COMPUTER SCIENCES CORP ARLINGTON VA

SOFTWARE PHENOMENOLOGY - WORKING PAPERS OF THE SOFTWARE LIFE CY--ETC(U)

AUG 77 B ELKINS, L HUNT

DAHC26-76-D-1006 AD-A053 014 NL UNCLASSIFIED OF 8 ADA 053014 器

しししににして

US ARMY INSTITUTE FOR RESEARCH IN
US ARMY INFORMATION AND COMPUTER SCIENCE
MANAGEMENT INFORMATION AND COMPUTER SCIENCE



DDC PROFINITE APR 10 1978

# SOFTWARE PHENOMENOLOGY

COPY AVAILATION PERMIT FULLI L

SOFTWARE LIFE CYCLE MANAGEMENT WORKSHOP

AIRLIE HOUSE

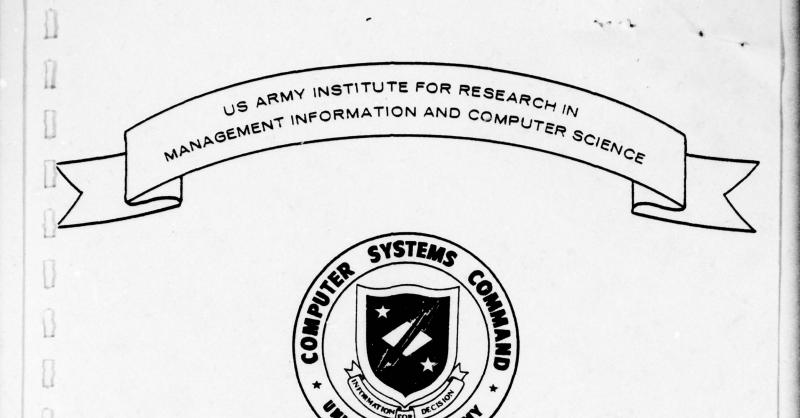
21-23 AUGUST 1977

NTEGRATED SOFTWARE RESEARCH

DISTRIBUTION STATEMENT &
Approved for public release
Distribution Unlimited

# **DISCLAIMER NOTICE**

THIS DOCUMENT IS BEST QUALITY PRACTICABLE. THE COPY FURNISHED TO DDC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.



# SOFTWARE PHENOMENOLOGY

WORKING PAPERS OF THE

SOFTWARE LIFE CYCLE MANAGEMENT WORKSHOP

AIRLIE HOUSE

21-23 AUGUST 1977



SECURITY CLASSIFICATION OF THIS PAGE (when data entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
. REPORT NUMBER	2. GOVT. ACCESSION NO.	3. RECEIPIENT'S CATALOG NUMBER
Software Phenomenology - Workin Papers of the Software Life Cycle	g	5. TYPE OF REPORT & PERIOD COVERE Symposium Papers
Management Workshop ) Airlie Ho	ouse, 21-23 August 19	FERFORMING ORG. REPORT NUMBE
. AUTHOR (s)		8. CONTRACT OR GRANT NUMBER (s)
Dr. Bryce Elkins Lois Hunt	15	DAHC 26-76-D-1006 NEW
PERFORMING ORGANIZATION NAME & ADDRESS Computer Sciences Corporation  400 Army Navy Drive	s	10. PROGRAM ELEMENT, PROJECT, TAS AREA & WORK UNIT NUMBERS
Arlington, Virginia 22202		12. REPORT DATE
Computer Systems Command	(12) (76)	23 Aug 77
United States Army Fort Belvoir, Virginia	Jerp.	13. NUMBER OF PAGES 684
4. MONITORING AGENCY NAME & ADDRESS (if diffe	erent from Controlling Office)	15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASS/DOWNGRADING SCHEDUL
DISTRIBUTION STATEMENT (of this report)  DISTRIBUTION STATEMENT A  Approved for public releases  Distribution Unlimited		
7 DISTRIBUTION STATEMENT (of the abstract entered	d in block 20, if different from re	port)
Software Phenomenology, Software Maintenance, Management Control Reliability, Human Factors, Individ	Life Cycle Managemer and Resource Allocation	on, Measurement of Quality an
ABSTRACT (continue on reverse side if necessary and Software Phenomenology presents the Management Workshop (August 1977) being executive summary with recommendant reports, a list of attendees, and	identify by block number) working papers submi y the participating aut dations, a summary o d the workshop schedu	tted to the Software Life Cycle hors. The volume also include f the plenary session with chai
Software Phenomenology, Software Maintenance, Management Control Reliability, Human Factors, Individual ABSTRACT (continue on reverse side if necessary and Software Phenomenology presents the Management Workshop (August 1977) be an executive summary with recommentmen's reports, a list of attendees, and	Life Cycle Managemer and Resource Allocation dual and Group Production of the Managemer and Group Production of the Managemer and Group Production of the Managemer and Ma	tted to the Software Life hors. The volume also f the plenary session wit le. The objective of the

SECURITY CLASSIFICATION OF THIS PAGE (when data entered) 20. ABSTRACT (cont.) problems, emphasizing short- and long-range approaches supported by quantitative studies.

#### PREFACE

The U.S. Army Computer Systems Command (USACSC) has been addressing Software Life Cycle Management (SLCM) problems emanating from current systems. These problems are likely to become even more significant because of the requirements for increasingly complex and costly software systems. Recently it has recognized that theoretical and experimental work in the area of software phenomenology is strongly related to SLCM. Since many separate researchers have been employing the global, phenomenological approach, it seemed appropriate and timely to bring them together to present their ideas, to engage in discussions, to establish any relationships that may exist between various concepts and techniques, and to recommend promising solutions and prospective research directions.

To achieve this forum, USACSC, through its Army Institute for Research in Management Information and Computer Science (AIRMICS) and the Integrated Software Research and Development (ISRAD) Program, decided to sponsor a workshop addressing the many facets of phenomenology and of software life cycle management. The workshop was to focus upon practical solutions to current SLCM problems, emphasizing short- and long-range approaches supported by quantitative studies. Its format was designed to encourage discussions of current problems and to elicit new perspectives on potential solutions through the interaction of researchers with varying views. The desirability of achieving practical results implied strong restrictions on the number of attendees, unavoidably excluding from attendance a number of researchers whose work would have contributed to the workshop's overall quality. It is hoped that their participation will be possible at subsequent workshops. Approximately 30 international experts in four general areas were invited to present and discuss papers and to recommend prospective research directions.

This volume presents the submissions to the Workshop provided by the individual authors. No attempt has been made by the editors to unify the styles or to referee the papers formally, but they have been ordered in what appears to be a fairly natural sequence. The volume also includes an executive summary with recommendations, a summary of

the plenary session with chairmen's reports, a list of attendees, and the actual workshop schedule.



# CONTENTS

Section		Page
	PREFACE	ii
I.	EXECUTIVE SUMMARY AND RECOMMENDATIONS	1
II.	TECHNICAL INTRODUCTION - M. M. Lehman	3
III.	PLENARY SESSION SUMMARY and Chairmen's Reports	9
IV.	AFTER DINNER ADDRESS On Software Engineering - John Staudhammer	29
v.	PAPERS PRESENTED	51
	<ol> <li>Software Maintenance and Life Cycle Management - Clair R. Miller</li></ol>	53
	2. Software Management Standards - Dennis W. Fife .	63
	3. Quantitative Evaluation of Software Quality - B. W. Boehm, J. R. Brown, and M. Lipow	81
	4. Management of Software Development - Edmund B. Daly	95
	5. Resource Estimation - Jules Schwartz	117
	6. The Cost of Developing Large-Scale Software - Ray W. Wolverton	131
	7. A Method of Programming Measurement and Estimation - C. E. Walston and C. P. Felix	153
	8. On the Relationships Between Design Theory and Software Life Cycle Management - Kenneth W. Kolence	175
	9. Beyond the Four Stages: What Next? - David G. Robinson	187
	10. Life Cycle Planning for a Large Mix of Commercial Systems - I. R. Elliott	203
	11. Project Life Cycle Modeling - Peter V. Norden .	217
	12. The Influence of the Time-Difficulty Factor in Large Scale Software Development - Lawrence H. Putnam	307

Section		Page
	13.	Evolution Dynamics-A Phenomenology of Software Maintenance - M. M. Lehman 313
	14.	Preliminary CCSS System Analysis: Using Techniques of Evolution Dynamics - M. M. Lehman and John K. Patterson 325
	15.	An Evolution Dynamics Model of Software Systems Development - R. S. Riordon 339
	16.	The Dynamics of Software Development I - George J. Fix
	17.	Software Complexity - L. A. Belady 371
	18.	Potential Impacts of Software Science on Software Life Cycle Management - M. H. Halstead
	19.	Notes on Software Reliability and Quality - R. C. McHenry
	20.	A Summary of Software Reliability Models and Measurement - Martin L. Shooman 419
	21.	Software Reliability Measurement - John D. Musa
	22.	Error Counting Models of Software Reliability: Some Comments, Criticisms and Proposals - F. N. Parr
	23.	Software Reliability Measurement: Some Criticisms and Suggestions - B. Littlewood 473
	24.	A Subjective Evaluation of Selected Program Development Tools - J. D. Aron 489
	25.	Applying SADT to Large System Problems - John W. Brackett and Clement L. McGowan 539
	26.	DOMONIC-A System to Document, Monitor, and Control Software Development - Dick B. Simmons . 553
	27.	Designing a Software Measurement Experiment - Victor R. Basili and Marvin V. Zelkowitz 579
	28.	Operational Aspects of a Software Measurement Facility - Marvin V. Zelkowitz and Victor R.

Section		rage
	29. Software Psychology: Shrinking Life-Cycle Costs - T. Love	605
	30. Computer Systems Engineering Management Education - John H. Manley	627
	31. A Theory of the Market Demand for Information Analysis Center Services - Peter G. Sassone .	633
	32. A Forum Approach to Software Development-The National Forum on Scientific and Technical Communication - Elizabeth Byrne Adams	649
VI.	LIST OF ATTENDEES	675
VII.	WORKSHOP SCHEDULE	681

#### EXECUTIVE SUMMARY

The Software Life Cycle Management Workshop was conceived and directed by Colonel Lawrence Putnam and Professor Meir Lehman. Through their efforts, 33 speakers and 17 observers gathered at the Airlie House in Warrenton, Virginia for a 2-day workshop addressing quantitative aspects of the software development process. The workshop was organized into four parallel groups addressing problems in the areas of Measurement of Quality and Reliability, Management Control and Resource Allocation, Phenonenology of Software Development and Maintenance, and Human Factors, Individual and Group Productivity Measures. The workshop opened with an introduction by Colonel Putnam, who stated the workshop objectives: (1) to identify problems; (2) to share ideas about promising solutions to the problems; and (3) to recommend prospective research directions leading to solutions. The technical introduction by Professor Lehman that followed set the scene for the two days of discussions, and its text is included in the proceedings. At the end of two days of parallel and general sessions, which included an after-dinner address by Dr. John Staudhammer entitled "On Software Engineering," a plenary session convened to summarize the results of each of the four sessions and to review the degree of success in achieving the workshop objectives. Recommendations made at the plenary session are listed below and are explained in greater detail in the plenary session summary.

#### Recommendations and Prospective Research Directions

Immediate emphasis should be given in the following areas:

- Standards for system design, documentation, process measure definition, measurement, and other software development processes
- Determining and defining <u>measurable attributes</u> of software quality and process productivity
- Data Collection and analysis as a continuing process
- Understanding discontinuities resulting from project size and complexity
- Flexibility as a system design attribute

- Design and construction of software modules for re-usability
- Refining configuration management to provide a simple, direct procedure for maintaining and controlling software system structure
- Verification and extension of techniques such as those arising from the
   Rayleigh curve cost model and from the results of the <u>Evolution Dynamics</u>

   Studies to establish their usefulness over a greater range of projects
- Implementation of <u>life cycle models</u> in the management control process
- Impact of environment, change and module integration upon software reliability
- Testing and using existing software reliability models.

## Future Research Recommendations

Future software research and development should be directed to the following areas, each of which promises added understanding and control over the software development process:

- System metrics
- System Evolution Dynamics and its interpretation
- Value assessment of proposed methodologies
- Development of tools to fill defined needs
- Maintainability (definitions, models, techniques and practices)
- System reliability (hardware, software, human)
- The implementation of change
- Software Science and Physics (in the sense of Halstead and Kolence).

#### Future Activity

At the conclusion of the plenary session, it was recommended that a second workshop be held to continue the discussions and to explore more deeply the phenomenology of software development.

# SLCM WORKSHOP-TECHNICAL INTRODUCTION M. M. LEHMAN

In presenting the technical introduction this Workshop on Software Life Cycle Management I should first try to explain why you have been invited and what we and the sponsors, AIRMICS and ISRAD, hope to achieve.

The meeting was conceived during a discussion between Colonel Putnam and myself at the 2nd International Conference on Software Engineering in San Francisco in October 1976. We each realized that, despite a difference of focus, his work on life cycle cost distribution was strongly related to work by Les Belady, Francis Parr and myself in Software Evolution Dynamics. The connection related both to the methodology used and to global conclusions reached. This same observation was made also by an independent observer who wrote in a review of the conference in the December 1976 issue of Computer, "Interpreting this result together with his earlier project estimation model, Putnam derived virtually the same management implications as those presented by Lehman: Large system development is inherently stable and will be driven to the natural parameters of the process; . . . the only real management options to influence the systems are Go, No-Go, Limit, and Freeze Decision."

In our brief talk, Larry and I observed that the technical basis of our common approach was phenomenology, the observation of the real world as it is. Measurement of its behavior, development of best-fit models that describe but do not explain that behavior, interpretation of the models, and the formulation and experimental confirmation, modification or rejection of theories will provide the basis of a software science and engineering discipline. The whole, of course, is a highly iterative procedure. I shall return to this in a moment, but first, let me continue my historical review. We also realized that there were other workers in Computer Science and Software Engineering adopting the same, or a related approach. Brooks in his Mythical Man-Month adopts a global view. Kolence and Halstead in what they have termed "software physics" and "software science" respectively, adopt the measurement, modeling and interpretation approach, as do people like Musa and Shooman in the reliability field. Many more

examples could be cited. We tried to invite such globalists to this meeting, but all could not attend. My apologies to those whom we were unable to locate.

So much as to the background. What is to be the subject matter of our discussions? As I have indicated, in the scientific sense our universe of discourse is the phenomenology of software development and maintenance. More pragmatically, we wish to direct attention to the meaning of, the need for, and the methodologies of Life Cycle Management of the software development and maintenance processes.

When questions of cost, throughput, reliability (or better, uncertainty) and malleability are addressed, one may not focus exclusively on one activity, on the execution time of one program or on the adaptability of the system at one moment in time. The life cycle of an application, of a program through its various releases, even of the process whereby an individual program is developed and maintained, is multi-faceted. There is continuous potential for tradeoff, of balance between local and global considerations, short-term and long-term implications. All program management occurs in an environment of continuous evolution of the application, the technologies and the system. Evolution is an intrinsic property of computer usage.

It follows that computer science activity must at least in part be based on observation, measurement and analysis of the total lifetime of systems and over systems. That is, de facto, there exists an approach to computer science and software engineering based on the concept and methodology of phenomenology.

Let me digress for a moment. Any engineering discipline is based on a combination of pragmatics, natural science and mathematics. In figure 1 I have tried to summarize my view of some major relationships between the real world, mathematics, science and engineering practice. In particular, I wish to draw attention to the major role played by measurement and phenomenology, particularly during the formative stages of a new discipline. I can do no better here than give you, in figure 2, two quotations that eloquently express and summarize what I have been trying to say.

#### NATURE OF ENGINEERING SCIENCES

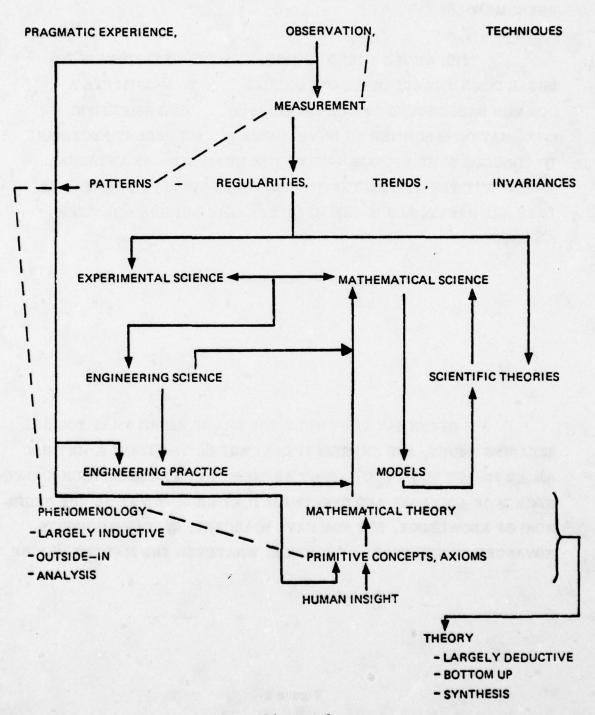


Figure 1

#### PHENOMENOLOGY

"THE WHOLE TREND OF MODERN SCIENTIFIC VIEWS IS TO BREAK DOWN THE SEPARATE CATEGORIES ..... TO SUBSTITUTE A COMMON BACKGROUND OF ALL EXPERIENCE .... OUR SCIENTIFIC INFORMATION IS SUMMED UP IN MEASURES .... WE FEEL IT NECESSARY TO CONCEDE SOME BACKGROUND TO THE MEASURES - AN EXTERNAL WORLD; BUT THE ATTRIBUTES OF THIS WORLD, EXCEPT IN SO FAR AS THEY ARE REFLECTED IN THE MEASURES, ARE OUTSIDE SCIENTIFIC SCRUTINY."

Eddington - Nature of the Physical World

#### MEASUREMENT

"I OFTEN SAY THAT WHEN YOU CAN MEASURE WHAT YOU ARE SPEAKING ABOUT, AND EXPRESS IT IN NUMBERS, YOU KNOW SOMETHING ABOUT IT; BUT WHEN YOU CANNOT EXPRESS IT IN NUMBERS, YOUR KNOWLEDGE IS OF A MEAGER AND UNSATISFACTORY KIND; IT MAY BE THE BEGINNING OF KNOWLEDGE, BUT YOU HAVE SCARCELY, IN YOUR THOUGHTS, ADVANCED TO THE STAGE OF SCIENCE, WHATEVER THE MATTER MAY BE."

Lord Kelvin

The term Computer Science has been around since the early sixities. "Software Engineering" was, I believe, first coined in connection with the 1968 NATO conference by that name. We may now legitimately ask--indeed, may have been asking for a number of years--whether these terms are indeed justified and what the science and the engineering disciplines really are. The pressure for measurement has been around in our community for a number of years, particularly but not exclusively in the area of performance evaluation. But the phenomenological investigations have been restricted to individuals working largely in isolation or in very small groups.

Our intentions were to identify as many of these individuals as we could and to bring them together to learn from each other; to challenge one another; to share experience, observation and conclusions; to identify and develop intersections and common viewpoints; to exchange tools and techniques; and hopefully, to extract some basic principles and generalizations.

These are worthy and ambitious aims in themselves. But under encouragement from our sponsor, we hope to go one step further. The meeting should seek to identify principles, methodologies, techniques and tools that are ready for exploitation. Additionally, we would hope to identify potentially fruitful research directions in software engineering, programming methodology, software management and in applied computer science. That is, we expect both to suggest areas and directions of investigation and research that will advance the development of computer science and software engineering as intellectual disciplines and also to develop concepts, methodologies, techniques and tools that are ready for applications; that will prove of significant value to the practitioners.

I have already taken up far too much of our very restricted time. Let us now get down to our business following the format that Colonel Putnam has outlined, but with willingness to change, to apply management feedback and control so as to maximize our collective intellectual gain and our real output.

#### PLENARY SESSION SUMMARY

CHAIRMAN:

Meir M. Lehman

PANELISTS:

Robert McHenry, Kenneth Kolence, Les Belady, Maurice Halstead

Manny Lehman pointed to the successful achievement of goals for the 1977 SLCM Workshop and the unanimous desire of participants that a second workship be convened early in 1978.

Each chairman was then asked to review activities during the technical sessions.

# Phenomenology of Software Development and Maintenance

Chairman: Les Belady

Les Belady presented his session's report, which was organized in top-down fashion (total system view, system structure/process, and methodologies), followed by conclusions and recommendations.

Each of the issues addressed within the session had a common objective to improve overall software management capabilities. Specifically, the discussants addressed three key management metrics -- predictability, controllability, and cost-effectiveness. It was agreed that a phenomenology of software development was needed and that its existence was evidenced throughout the workshop discussions.

<u>Total system view of software</u> - A total system view of software was stressed, and observations of software development based upon the following three perspectives were discussed:

- the stages in user growth
- the total life cycle of a software development project
- the single release process

It was noteworthy that each perspective was supported by observed data on actual system developments.

System structure/process - To achieve the controlled development of software systems, the properties unique to the system structure must be identified, the totality of interrelationships (hardware, software, human) must be thoroughly understood, and full use of all potential feedback processes must be made. Knowledge in these areas is necessary to enable the system content, structure and development process to change as the system requirements change.

Methodology - To aid in the software development process, common standards must be established for clear, unequivocal communications links between the various groups involved.

General procedures which promote greater management visibility and control must be established throughout the software development process. Such overall procedures can be supplemented by project specific procedures as required. Tools which support all phases of the software development process should be developed and made easily accessible. One attempt was made to assess the value of a number of software development tools and methodologies based upon their implementation cost and value to the overall development process. Explicit tests of other proposed tools should be conducted, so that their value to the life cycle process can be determined.

Conclusions and recommendations generated during the session are listed below:

#### Conclusions -

- The software development process can be described by a (partially known) measurable, generic set of phenomena.
- Each project requires a management system specific to its environment,
   which is based upon the generic phenomena and uses locally generated
   measure values for planning and project control.

 Any organization can adopt and continually improve uniform procedures and methodologies which fit all their projects and systematically apply the generic phenomena within the management system selected.

Recommendations - The recommendations set forth by the session are divided into two sets -- those pertinent to "Problem Management" and those pertinent to "Problem Avoidance."

## Problem Management -

- <u>Data collection and analysis</u> should be a continuous process associated with any software project and should be reported to the software development community for general use.
- <u>Standards</u> for system design, documentation, measurement and other processes associated with software development should be promulgated.
- Configuration management must be studied and refined to provide a simple, direct procedure to maintain and control software system plans.
- Education is a necessary ingredient in the successful software development process.

#### Problem Avoidance -

- <u>Size/complexity</u> of projects have been identified as key parameters in determining overall management procedures necessary for successful project control. An observable discontinuity in these two parameters requires greater understanding. Considerable research is required to clearly establish these parameters, their range, and their relationship to the overall project.
- Changeability must be carefully designed into systems in order to allow for the unavoidable user requirements shifts and management-determined redirections during the life cycle of a software system.

Reusability of software modules should become an established practice.
 This includes the development of software libraries and a communications mechanism to assure that such software libraries are fully exploited by system developers.

<u>Recommended Research</u> - The session recommended four specific areas for future software research and development:

- System metrics
- System invariants or regularities
- Value assessment of proposed methodologies
- Development of tools to fill defined needs.

## Management Control and Resource Allocation

Chairman: Kenneth W. Kolence

The working group had 17 people or more in attendance at each of the working sessions, of whom 7 were "observers" and 10 had been invited to present papers. Following the normal procedures, each of the 10 (with one exception) gave a 10-15 minute brief overview of their papers. The exception was the session chairman, K. W. Kolence, who felt his paper was not worthwhile presenting due to both a shortage of time and the fact the group had already focused on the project life cycle curves as the primary topic to be considered. A list of the papers and presenters at the first two sessions is included as an appendix to this report.

There were three people whose papers directly addressed the use of Rayleigh curves as a model of the manpower-time relationships in design and related project efforts. Of these, the papers of Larry Putnam and Peter Norden were heavily oriented toward explanation of observed empirical data in terms of the Rayleigh equations. J. Spruce Riordon's paper proposed a process control model of the equation to investigate its dynamic characteristics and represents an extension of the work of Belady and Lehman.

In addition to the above-mentioned papers, those of Ian Elliott, Clair Miller, and Jules Schwartz also addressed the basic questions related to the use of Rayleigh curves in practice. In particular, the constraints on available manpower and management allocation practices are the topics of the latter two papers. Elliott's paper concerns the implications arising from the overall age of code within the life cycle on the manning levels within a programming organization. All three of these papers relate directly to the Rayleigh curve papers in at least some way, so the majority of the basic group of 10 papers were relatable. Each of the other four papers were on different topics, but again generally relatable to the topic of Rayleigh curves.

Norden, Putnam, and Riordon were requested by the group to give detailed presentations of their papers, which they did in the order given above. It was the clear consensus of the group that Norden's and Putnam's presentations were convincing evidence that Rayleigh curves were empirically shown to represent the manpower-time relationships in design projects. The research recommendations of the group were therefore based on the belief that a fundamentally important empirical relationship had been established for the software life cycle concept. However, a variety of important anomalies and discrepancies relative to the real world were commented on, and need to be taken into serious consideration in adapting these curves for general software engineering use. These are discussed below, in no particular order of importance. Many of these were commented on in the group discussions, but some are the result of subsequent reading and considerations by the session chairman. Those in the latter category are clearly identified.

In the presentation by Norden, the Rayleigh curves were empirically and logically shown to be valid only for each activity which coherently reflects a process of making a large set of decisions. For example, a requirements phase, a design phase, a prototype development phase (for equipment) etc. The combination of phases into a single overal activity did not form a Rayleigh curve, nor logically could it. Putnam's empirical data for major software development projects showed the overall activity was reflected by a Rayleigh curve. The strong implication here is that the current division of a software

life cycle into individual phases is not being implemented in practice in a way which represents separate decision making activities. That is, the specification-design-implementation techniques in use do not reflect actually distinct processes of decision making; e.g., the coding process includes elements of the specification process. If this is so, and clearly practice would support this thesis, then a rather thorough reexamination of current "software design methodologies" is obviously needed to avoid "code-level specification decisions." This is perhaps the most fundamental anomaly which was brought to the group's attention.

A second set of problems of practical importance was also discussed. The Rayleigh curves have been used successfully to predict the total manpower needs and elapsed time of a decision-making phase (e.g., a full software development cycle). The two parameters needed are 1) the total manpower expenditure K, and 2) the curve "shape parameter a, which represents the peak manpower loading. Given these, tracking of actuals to planned and fitting early actual data to the Rayleigh curve equation can project actual total manpower, peak loadings, and total elapsed time with good accuracy. Alternately, given a project with a good estimate of the total effort required and the schedule objectives for completion, the manpower loading necessary can be determined. Further, Putnam's work shows that certain regions of the manpower-time planes are "forbidden" in the sense that actual projects cannot reflect these combinations. In other words, a thousand man-month effort cannot be completed in one month by assigning a thousand people to it. Putnam has interpreted this to mean that systems have an inherent "degree of difficulty" which severely constrains the possible manpower-completion time parameters of any given software project.

The problems which arise from these considerations are both practical and theoretical. On the practical side, use of these curves to originally project the time required to complete an effort depends critically on the estimated man-months. Reasonably accurate man-month estimates can be made a priori only when the group responsible has sufficient previous experience with similar projects. New types of efforts by an organization (e.g., an interactive online system as opposed to batch systems) cannot usually be

estimated with any reasonable accuracy. In other words, if you've done it before, you know how long it will take. If you haven't, you don't. In addition, there are clearly learning curve variations in the effort required.

On the theoretical side, the use of the quantity K (total man-months) as a variable to represent the "complexity" of a design effort is untenable on a variety of grounds. Fundamentally, two things are being represented by K: 1) the design complexity, i.e., the number of, and perhaps interactions between, the design choices, and 2) the ability of the individuals involved to recognize the need for these choices and the rate of correctly making them. The variable K combines these, whereas for predictive purposes one would like them separated. This would appear (to the session chairman) to be a fruitful practical research area, as it would lead to a better understanding of the practical meaning of the term "design complexity," and a means of evaluating the relative capabilities of different design teams to successfully complete an effort. It may also provide a means for evaluating the effectiveness of different "design methodologies," as was noted in the group when Norden presented some comparative data using different methodologies.

A second theoretical question with significant practical importance is the understanding of the term "similar designs." In estimating K, this is an important consideration. It likewise is of importance in understanding the theoretical term "design complexity." The group discussed this question briefly, and a general consensus existed that a taxonomy of software systems was of considerable practical value. From other considerations (see K. W. Kolence's paper for the workshop), the session chairman believes this taxonomy should be based purely on structure, not function. Further, any research directed to this goal should be fully cognizant of the many uses to which a successful taxonomy would be put.

Although the topic was not fully discussed within the group, it is the session chairman's belief that Dr. Maurice Halstead's "software science" work and the work by Belady and

Lehman on the dynamics of evolution of software are sufficiently supported with empirical data to be seriously considered in any research on the general topic of Rayleigh curves. They address similar questions, and appear applicable to as wide a variety of problem areas. If all these approaches continue to be supported by empirical data, then a strong presumption exists that a synthesis may well produce impressive and immediately practical results.

A third set of anomalies and, in the opinion of the session chairman, weaknesses exist in the material presented by Putnam, though not Norden. In a sense they appear trivial, and in a practical sense they truly are for current use of Putnam's ideas. In a more abstract sense, however, they are important if Putnam's work is to be extended and integrated with other empirically based organized knowledge such as Kolence's software physics and Halstead's software science. The problem is the use of certain terms and, more especially, a consistent use of appropriate dimensional units. (This criticism applies equally, in the session chairman's opinion only, to Halstead's work.) Putnam says the factor K + t has units of power (work per unit time) and K + t, units of force. This is an incompatible statement. Putnam also says that a system has an "inherent difficulty factor" represented by  $K \div t_d^2$  and at the same time says that this appears to change somewhat with what group is performing the work. In essence, this was discussed earlier in the report, but the point being made is that the statement and the name "difficulty factor" are incompatible. Again, strictly in the session chairman's opinion, a more rigorous approach to accuracy in statements and dimensions is needed in this work now, before this material is spread too widely and these deficiencies can no longer be corrected.

The other major point considered by the group was the question of dynamic behavior analyses of the Rayleigh equation in its present form. Before turning to it, however, it should be emphatically stated that the criticisms above should be considered small in comparison to the stature of the actual achievements presented by Putnam and Norden. It was the group consensus that the work was outstanding and of immediate practical

value in current efforts. This was reflected in the major recommendations of the full group to the workshop.

Spruce Riordon's paper on the dynamic behavior of a system with the Rayleigh equation characteristics was well-received by the group. Due to a shortage of time, there was little in the way of a critical discussion on the subject matter. Within these limits, however, the group clearly felt that the dynamics and/or transient behavior of Rayleigh curve systems should be investigated. The difficulty of course would be how to gather empirical data to validate any mathematical results that are obtained though the Belady-Lehman work may represent a significant step in this direction. The group consensus was that such investigations represent a potentially fruitful research area. However, the personal opinion of the session chairman is that this will only be true if careful attention is paid to the problems of empirical verification from the very outset of the effort.

Rayleigh curves probably do represent a "natural" behavior of a design group on a design problem, and of similar groups with similar problems. That is, given a large number of people assignable to the task, the amount of effort actually required and the time span of such effort does appear to reflect some natural laws of design and the design process. As pointed out by Elliott, Miller, and Schwartz, the real world often, if not normally, does not have a large number of assignable people for a task. As a result, manpower loadings are often flat or trapezoidal, and may or may not be sensitive to the "natural" loading requirements of the effort.

While a fair amount of discussion was held on this subject within the group, little in the way of positive suggestions emerged. In the session chairman's opinion, this may have been due to the fact that we failed to ask the right questions; e.g., in a constrained world, what are the best manpower strategies to obtain the best productivity from a limited number of people. This of course may not be a "right question" either, but it certainly would have led to a more productive discussion. The problem should be faced more directly than has apparently been the case from the data and presentations available, and again may well represent a fruitful research direction.

The recommendations of the group were proposed and discussed within an hour's time, and so cannot be presented as carefully weighed and precisely stated. Indeed, much of the Session Report discussion has been for the purpose of placing these recommendations in a proper context. I have taken the liberty of injecting some personal opinions and comments into that discussion, but only when these points had not been brought out in group discussion due to the limited time available for study and reflection.

Even with a recognition of the somewhat hasty nature of the group efforts, it should be recognized that a remarkable unanimity of opinion existed on the value of Putnam's and Norden's work, and on the need for efforts such as proposed by Riordon. As a result, the recommendations given below should be considered as important general statements, arguable perhaps in the letter but not in the spirit of what they say.

The general recommendations of the group were as follows:

- 1. The group recommends that the Army Computer Systems Command use the Rayleigh curve techniques as developed by Putnam, Norden, and others as a predictive and control tool for software systems development. In particular, it is recommended that a continuing test of the validity of these techniques in CSC projects be made over as wide a range of project size and content as possible. At the same time, other useful predictive techniques such as those of Belady and Lehman should be similarly applied, evaluated and developed.
- 2. The group recommends research into the underlying "phenomenology" or "physics" of the processes and structures which give rise to the observed behavior. The purpose of this research is to understand the causes and possible trade-offs in design complexity, manpower, and time-to-complete such efforts so as to improve the process. Where possible, such research should be compatible with other empirically based "phenomenology." Both static and dynamic system behavior should be studied.

- 3. The group recommends that the Army Computer Systems Command begin to develop methods and procedures to implement presently available life cycle curve knowledge and understanding for purposes of both original estimates on project manpower and completion times and for monitoring actual progress. It is suggested these be tested on several projects for accuracy and appropriateness, simplified where feasible, and integrated into the appropriate labor and/or cost collection systems for routine use in subsequent projects.
- 4. The group believes many other potential and practical applications of these concepts may exist, especially when combined with other emergent "phenomenologies." As such, it recommends appropriate research and consideration of these concepts in other problem areas of the overall life cycle management process.

# Measurement of Quality and Reliability

Chairman: Robert McHenry

Robert McHenry presented the results of his session in the form of six specific recommendations. These were to collect data, use and test existing models, characterize the problem environment, characterize the effects of change, characterize integration effects, and characterize quality. In addition, topics were noted which should receive attention at subsequent forums addressing SLCM problems. Details of these recommendations follow.

# Recommendation 1: Collect Data

Impose mandatory data collection on Army in-house and contracted software projects to support intra-project reliability analysis and reliability research.

As a prerequisite to large-scale, mandatory data collection, clear definitions of all data items must be developed. First-cut definitions appear in recent reports prepared for RADC. The participants identified the following data requirements:

- "Formal" reporting, i.e., failures found in baselined products requiring formal problem reporting applicable to operation as well as development and test
  - calendar time
  - execution interval
  - process phase (requirements, operation, etc.)
  - priority or severity
  - qualitative description (taxonomy)
    - detection (e.g., review, inspection, execution, etc.)
    - failure
    - code
  - was failure induced as result of correction
  - did failure uncover similar errors (how many)
  - resource consumption: labor, computer
    - · detection, e.g. test team
    - diagnosis (hardware vs. software vs. person)
    - correction
  - identification of modules fixed
  - if closed -- when fixed or decision not to fix
  - Note: open failure maintained if associated errors cannot be diagnosed
- · Equivalent information for changes to baselined products
- "Informal" reporting, i.e., failures found in unbaselined products (and not subject to problem/change control in CM/QA terms), e.g., library updates may estimate failures.

# Recommendation 2: Use and Test Existing Models

There already exist several interesting models for software reliability measurement.

These should be used and tested against historical and current project data. Future work should concentrate on achieving the following properties:

Simplicity

- Informativeness
- Appropriateness
- Practicality
- · Wide applicability.

This might be done by addressing the following questions:

- What failure models are "natural" for software?
- How can "costs" and "utilities" be incorporated to achieve specified reliability levels?
- How do we integrate hardware and software reliability models into a model for system reliability?
- Can knowledge of program structure be used (e.g., modules)?
- What are the common properties and differences of existing models?

The state of the art is such that no models can be definitely chosen in preference to others. Parallel and competing work in the above areas is desirable.

# Recommendation 3: Characterize Environment

Initiate research studies to investigate the characterization of the input space of a program and how changes in the input space can affect the operational reliability of the program.

To improve reliability modelling and reliability prediction by characterizing features of execution environments which influence the observed rate of failure.

Develop automatic recording to be built into future software so that the type of usage causing failure can be identified.

Identify some qualities in the execution environment which affect the observed rate of failure of a software system. It should be used as inputs for a prediction of the relative failure rates of different test/execution regimes.

Current reliability models assume with little justification that the reliability information collected before a system goes live can be used to predict subsequent operational performance.

Develop a characterization of the components of stress than can be applied in testing software, e.g.:

- Input stress rate, volume
- Algorithm stress near-singular cases
- Reduction stress singular cases, input errors.

Develop strategies for software stress testing.

## Recommendation 4: Characterize Change

Determine how changes are reflected in the reliability models. Identify rate of change metrics which can be used to describe the nonsmoothing aspects of reliability change.

Software changes reflect changes in requirements/specifications. These changes might be treated in various ways:

- May be handled quite adequately by simply continuing to use existing models
- May imply a new program on which reliability estimates must be made
- May imply restarting the testing process and generating reliability data.

The questions concerning absorbing new data by already existing probability distributions are:

- Are there specific discontinuity points at which such absorption is reasonable?
- Can we treat such distributions as a collection of distributions?

The characterizing of change is important to change introduction strategies to assess the extent of reasonable change and to determine blocking strategies for program releases.

#### Recommendation 5: Characterize Integration

Determine the effect of integrating modules on existing reliability models which treat only one program (module or system).

Collect data on interface errors and determine whether errors are intramodule or interface errors. The latter may include all combinations of software-hardware-human interfaces.

Developed models which relate software error incidence to cost of integrating previously-tested and new software.

## Recommendation 6: Characterize Quality

Determine the measurable attributes of software quality.

Traditional quality assurance may be paraphrased as conformity assurance. Specification conformity is not adequate to assess several software characteristics associated with quality such as maintainability and portability. The quality characterization research includes specification and realization techniques in addition to attribute definition. Some first-cut definitions and approaches have been prepared under government contracts.

## Additional Topics

Several topics of importance to this group were not discussed because of lack of time. Foremost among these topics were:

- · Maintainability techniques and practices
- · Maintainability definitions
- Maintainability mathematical models
- · Availability definitions
- · Availability mathematical models
- System reliability including hardware, software and human operator
- Acceptance tests as stand-alone procedures, and as they pertain to reliability models.

The reader should not infer that there is a lack of preliminary work or increase of importance in these areas. In fact, should this discussion resume in the future, such topics would be first on the agenda. Continuing work in these areas is desirable.

Human Factors, Individual and Group Productivity Measures

Chairman: Professor Maurice Halstead

## Session Overview

In an industry as labor-intensive as our own, the measurement of both individual and group productivity is a vital concern. The application of knowledge gained from these measures to real-world problems is certainly of equal concern. The participants in the "Human Factors, Individual and Group Productivity Measures" sessions dealt with the following general areas of interest:

- The basis for Software Engineering
- Measures of productivity-metrics
- Measurement techniques
- · Application of measurement results.

One of the major outcomes of this session was a realization of the synergistic relationship of each participant's area of endeavor to those of the other participants. For example, experimental data collected by Dr. Love and actual project data collected by Dr. Walston were further supporting evidence for Dr. Halstead's Software Science.

The sessions were quite orderly and, for the most part, consisted of the presentation of the SLCM papers with interaction among the participants from time to time on specific issues raised by the presentation. The meat of the presentations can be found in the papers themselves. What follows is a very brief summary of each presentation and a final section listing the problem areas, promising research areas, and research objectives identified, or more appropriately distilled, by the participants during the final session.

#### **Presentation Summaries**

Dr. Halstead

Dr. Halstead's presentation, "Potential Impacts of Software Science on Software Life Cycle Management," pointed out that Software Engineering must have a natural science base to be a "quasi-complete" engineering discipline. Halstead's "Software Science" might be considered as one candidate. "Software Science" is based on a few language independent parameters which can be easily (even automatically) measured. The derivation of Halstead's E was shown along with supporting evidence of its accuracy. A number of applications concerning software life cycle management were mentioned and specific areas for study indicated.

#### Dr. Love

Dr. Love's presentation, "Software Psychology: Shrinking Life Cycle Costs," also states that Software Engineering must have an underlying body of scientific knowledge and that the disciplines of psychology and statistics may help add to that knowledge. To understand and successfully apply any new software engineering technique, we must know why it is better than another. Software Psychology is the "application of the knowledge and technology of psychology to improve the productivity of programmers."

The results of three completed studies were presented. The first study represents the first time Halstead's theory of Software Science was tested with appropriate experimental controls. A high correlation was found between performance in the experiment and Halstead's E. Other studies "Individual Differences in Programmer Productivity" and "Critical Factors in Software Development" provided some interesting and some counterintuitive results.

Some expected results of this work:

- A set of equations to help evaluate actual techniques relative to schedule, cost and productivity
- The relationships among techniques
- The identification of new techniques which may be of use.

#### Dr. Manley

Dr. Manley's presentation, "Computer Systems Engineering Management Education," concerned the education of practitioners of software engineering. The premise that

industry roles are changing and traditional educational systems no longer meet their objectives indicates a new approach is required. Johns Hopkins' approach involves the blending of science and management to provide a realistic methodology for not only dealing with but improving the management world.

Dr. Manley's paper gives a complete overview of his approach and complete descriptions of the Computer Systems Engineering Management courses offered at Johns Hopkins/APL.

Dr. Walston

Dr. Walston's presentation was based on "A Method of Programming and Estimation" by E. E. Walston and C. P. Felix published in IBM Systems Journal, Vol 16, No. 1. The motivation behind the establishment of the programming measurement data base at IBM's Federal System Division, the vehicles used to take measures, and current and ongoing work were presented.

The data base is used to evaluate new hypotheses and improved programming techniques and to provide indirect feedback to the data supplier as well as to provide a historical record of software development performed.

The actual measurement program generally starts at contract award and concludes at acceptance or cutover, i.e., the development phase only. The method of collecting data was discussed and some preliminary results of measurement techniques were presented.

Dr. Walston showed some recent analytical results with the data in his article supporting Dr. Halstead's E Hypothesis.

#### Raymond Wolverton

Mr. Wolverton's presentation, "The Cost of Developing Large-Scale Software" described a "state-of-the-art," accurate (n 10 percent accuracy), working cost estimation algorithm usable by end management. The methodology based on a 25x7 structural forecast matrix (25 activities x 7 phases) was presented.

Actual project experience was also discussed and a number of interesting units of measure were discussed:

- The number of cards a programmer can maintain is a more meaningful measure than the cost to change a line of code during the maintenance phase
- Basic unit of measure is 1000 machine language instructions
- Errors counted per month per 1000 instructions.

#### Dr. Zelkowitz

Dr. Zelkowitz presented a short overview of the University of Maryland/NASA Goddard Space Flight Center Software Engineering Laboratory. The general goals of the facility are to monitor and analyze the production of software at Goddard. Specific goals are to organize a data base, isolate parameters and find out what is happening as opposed to what should be happening during software production.

Three types of experiments are being conducted: <u>screening</u> - to discover the range of variables (to debug methodology); <u>semicontrolled</u> - extension of screening methodology with more control; and <u>controlled</u> - a further extension of service-controlled (use of duplicate projects).

The problem of consistency across projects (transferable data) was discussed. This is a general problem. Among the session participants one counts lines of code (JC L and comments included), another counts machine language instructions, and a third counts executable FORTRAN statements.

## Key Items

## Problem Areas

- A standard nomenclature is needed -- both a basic vocabulary and a set of common definitions is a must
- A general need for more data -- in particular data that is transportable.

## Promising Experiemental and Theoretical Research Areas

- · The implementation of change
- Computer Systems Management Engineering
  - Management Science
  - Behavorial Science
  - Team Organization
  - Practical Applications
- Software science.

## Research Objectives

- Establishment of a "quasi-complete" engineering science base
- Design of experiments
- Identification of erroneous conclusions due to incorrect parameters.

# ON SOFTWARE ENGINEERING

After Dinner Address

presented by

Dr. John Staudhammer

On Software Engineering

John Staudhammer\*

US Army Computer Systems Command
Fort Belvoir, Va.

Over the last fourteen months I served as Technical Advisor to the Army Computer Systems Command. During this time I had occasion to examine a great many problems in the production of responsive software and to discuss with the Commander, Major General Jack L. Hancock, his views on software engineering. This presentation borrows heavily from his ideas and his published speeches (1).

One of the functions of the Technical Advisor was to chair the Army's Integrated Software Research and Development (ISRAD) Program council. During this time ISRAD was supported by a contract, competitively won by Computer Sciences Corporation. Two major significant actions were completed with this support: the hosting of the Army Software symposium of May 1977 (2), and the convening of this workshop.

ISRAD has also become a forum for ideas in software engineering problems, the most pressing of which is the adequate prediction and management of costs. With Colonel Putnam's work on the dynamics of software life cycle costs (3,4) we feel that we have an understanding of the enormity of this problem and we are conviced of the need to further explore and find the major factors in this matter. We are confident that better methods will emerge in the near future.

The Army Computer Systems Command comprises about 1500 people and all aspects of software engineering appear in the command's daily

<sup>\*</sup> On leave from North Carolina State University.

operations. The command is the developer, maintainer and disseminator of all financial, personnel and logistics programs in use at all Army installations. We do not operate the software, but are responsible for delivering operable systems and must respond to emergencies that may arise in executing our codes. To insure uniformity all of the systems software is distributed in object code form to about 100 installations.

The work of the command is in creating large business codes. The average program size is 300,000 lines of COBOL code, has a life of some seven years and costs about 350 man-years. The design-time, the time for first release after official funding, is 2.7 years. Our average coding rate is 2000 statements per year, giving a gross figure of about \$20 per statement, allowing a 100% overhead. Clearly this is a large operation and a systematic design procedure is mandatory.

One of the first actions I was involved in last summer was the selection of a command-standard procedure for program design. After a study of existing methods we chose the Jourdan-Constantine structured design methodology. Since not all 500 programers could switch to this method overnight, a phased implementation plan was agreed upon; also, we decided to have two pilot projects to demonstrate the efficacy of the method.

One, Project VIC-Visibility of In-transit Cargo (5) —was done in relative isolation in Europe. This system allows the tracing, routing and re-routing of various cargo units throughout a theater of operations. The original user-supplied requirements were carefully reviewed prior to code design. At that time the required effort to first fielding the code

was estimated at 5000 man-hours. Since the effort was a relatively modest one it was felt that little risk was involved in trying the newly decided-upon Structured Design Technology. Throughout the design little direct effort was made to gather special data on the performance of the design team. With the user working in daily contact with the program designers the Detailed Functional Requirements were analyzed and the top-down design was started. Quickly it was apparent to the user-team as well as the code-developer team that the stated requirements were grossly inadequate. Of course, both parts of the team had to have the same background training and had to use the same tools.

With the inadequacy of the requirements accepted by the user, the effort required was re-estimated. It is now at about 15 man-years, some six times the original guess. The tools used relied on a more knowledge-able user and requirements more carefully analyzed. It is also worth noting that the user continues to have a requirements analysis group about as large as the design team. The net result so far has been a more carefully thought—through design, a user who now has far more detailed insight into his own business and a code design understandable by the eventual user. Aspects of the system have been uncovered that were unsuspected earlier in this work. For example, security aspects were discovered early in the design and steps for adequate safeguarding could be initiated early; without the systematic design process, such surprises often necessitate massive code changes costing sometimes as much as the original "complete" code.

The other project, the Direct Support Standard Supply System (DS-4) (6,7) was done with much forethought to gathering statistics on the

progress of the design team so that the efficiency of using the structured design process could be evaluated. Compared with the VIC project the design was conducted in the limelight. This project was a re-direction of an earlier design effort which was given added and requirements started anew in June 1976. The target date for prototype code was Summer 1977. Using historical data on similar projects not done in a structured way Colonel Putnam estimated the coding project to yield 30,000 lines of COBOL code, and using his risk analysis procedures he estimated the chances of successful completion at about one percent. He also gave 50-50 odds for completion in 18 months. The latest data indicate that the program now has about 3000 lines of Meta-COBOL, completed about 1 June 1977, with "most" of the system executed to completion at about 10 August and that the overall program will contain about 30,000 lines of Meta-COBOL. The design effort has been about 15 man-years so far and may go to 20 man-years.

There was an additional requirement placed on this system: although developed on an IBM 370 system, targeted to run on an IBM 360/50 machine, there was a requirement to demonstrate it on a minicomputer. Several approaches to this conversion were taken; one of these converted about 10,000 lines of COBOL (obtained by expanding the Meta-COBOL source) in about 600 man-hours on to a PDP-11/70 system. While a few compromises had to be made, basically programs and files could be converted with relative ease once the functions of the code were well understood (8). The presence of the lead programer of the original code design team on the conversion team was of course a great help, but probably more

important was the enforced discipline of structured design concepts, of meaningful mnemonics, adherence to ANSI standard COBOL in the original source and readable code. However, the single most important factor in this conversion was the high motivation of the team members.

The basic conclusions on the DS-4 exercise so far are that code conversion productivity is highly dependent upon motivation; that there is a great need for functional expertise on the conversion team, and that code production rates are about the same in structured design as in more conventional methods. There is, however, a marked difference in the ease with which structured code can be read; this indicates a great hope for reduced maintenance costs.

Although data is scanty, the code production rates for COBOL and Meta-COBOL seem to be about the same. This in turn would indicate that efficiencies in creating a running code are most likely to come from changes in source language rather than enforcing structured constructs using a given language.

Part of the problem in comparing two higher-order languages is the very word "productivity". Basically what is being measured is not clear - for example, code produced with many inefficient statements could well result in more lines of code, but may require less effort, less design, for its creation. By the conventional measures of "code productivity" the inefficient program would be judged a better product. Probably the only true measure of a programer's effectiveness is the rate at which he reduces system requirements to checked-out code. But since these

requirements are all quite different, no good objective measures exist for measuring programer effectiveness. This of course should not be a surprise, for there are no objective measures for creative activities in any field. Recognizing this fact we should not delude ourselves by citing code production numbers a measures of effectiveness. Possibly software quality measures eventually may shed some light on this subject.

The problem of code portability is not unique to the Computer Systems Command. We do have, however, a peculiar environment where we are forced to produce and maintain software systems which will concurrently be executed on various alien mainframes. Part of the problem is to deliver checked-out codes to dissimilar equipments. One project developed a single source code, maintains that one source and produces from it different target object codes. In this case IBM 370 code was in existence and undergoing prototype evaluation when an additional requirement arose to transport the software to selected other large machines. Because the system was still in development, it was subject to substantial changes to satisfy new or revised functional requirements. Demands for final systems availability of all versions required concurrent parallel efforts - that of continuing the basic system development and modification - and that of developing two additional versions for CDC 6500/6600 and Univac 1106/1108. A team developed the new versions while the original Application System Developer (ASD) continued his work on the prototype. A copy of the current IBM baseline was transferred to the team for its use. Any subsequent changes made to the baseline by the developers were uniquely identified in the source data sets.

Following the initial multi-version development and validation of all three hardware versions against the initial baseline, the multi-version source programs were updated with the system changes which had been implemented by the developer and again validated. The multi-version system software package was then placed into continuing mainteneance (9,10).

Both the Structured Design and the single-source, multi-target code are efforts to anticipate changes in the way we will have to do business in the future: we foresee a multiplicity of hardware which must be accommodated in the various Army installations.

One of the research areas of deep concern is the assessment of factors in portability of software, particularly large COBOL source code. It is well understood that language standards, implementation differences and degree of adherence to structured design constructs all have a great impact on the ease with which programs can be transferred. A recent study focused on these and related questions in the framework of current practices (11). While questions of data portability have not been resolved, the recommendations include using only a subset of ANSI COBOL consistent with the implementations on all target machines, the use of a translator to replace vendor-dependent syntax, specialized Meta-COBOL macros which will expand to specific vendors' language constructs and the avoidance of specialized executive software.

This effort was a recognition on this agency's part that systematic research and development work had to be undertaken in all aspects of software engineering to aid in developing methods for more responsive

software. A comprehensive five-year research and development plan was formulated and published (12). It is intended that this plan be updated and modified as necessary and thus to be made into a "living document". Several hundred copies have been distributed to other Department of Defense agencies, industry and universities; it is hoped that this plan will form the basis of research proposals as well as proposals for changes of the plan itself.

While we are waiting for all the steps in the establishment of far more effective software engineering methods to be researched, evaluated and implemented, we may outline some of the more likely future solutions. One of our most serious shortcomings is the inability to forecast development efforts and another is the failure to produce satisfactory software. Generally more and more system failures are being attributed to the fact that the potential system user does not state his requirements clearly, adequately, in sufficient detail, in an understandable language, so that the system may be properly designed. The literature has pondered this problem and the general thrust has been twofold. The first thrust is for tools that designers can use to compensate for poor user requirements, and the second is aimed at how to get the user to specify his needs in a "clear, consistent and unambiguous statement which can serve as a basis for the follow-on design specification."

However, tools to compensate for poor user requirements do not substitute for having the user have a clear understanding of his own needs. For twenty-five years we have been after the user to specify his requirements in a complete, consistent and unambiguous manner without success. We must therefore conclude that in many cases it is impossible for him to do so. Providing only us, the program developers, with better requirements analysis tools is only a minor help. In the design of management information systems we must be responsive to the manager's needs. However, management decision making is not a science, but an art. In fact, it is, as one author stated, not one art, but two: the art of making decisions based on inadequate information, and the art of living with the results of those decisions.

Management is managing change. The abnormal situation is the status quo which the manager has to overcome in order to get back to the normal task of causing and manging change. If one can identify the decisions that a manager has to make, identify the criteria by which he makes those decisions, put a value on the resulting courses of action, and hold this value system constant, there would be little need for highly trained and qualified managers. In fact, we may require fewer computers as well. But mangers can not identify the decisions which they will have to make, can not identify the criteria they use, can not quantify the criteria, and can not put a value on the resultant courses of action; and they can certainly not express them in "a complete, consistent and unambiguous requirements statement which can serve as the basis for design specifications". And above all, there is no way that mangers can decree holding requirements constant.

In asking a manger to define his requirements, we must recognize that he can respond only with experiences and knowledge which he already has. Since the system he is asking us to develop is to operate in the future we must introduce him to data processing ideas, solutions and techniques which are likely to be used in the future, and must introduce these ideas to him during the process of requirements definition so that he may make reasonable choices about his evolving system. We are somewhat in the role of a restaurant specializing in Antarctic food: although we may have a menu, its entries are likely to be meaningless to our potential customers, they are unlikely to know the taste characteristics and we will have to be ready to describe, explain and demonstrate on occasions. The same ought to be true for the process of devising new computer-based systems. But the almost universal mode of systems design is to demand of the user his requirements in a format which is complete, not over-constraining on we, the developers, but which is clear, concise and unambiguous. And we want those requirements on time. Also under no circumstances must the user tell us how to do the job. Straightjacketing him to such an extent is problem number one.

All of us are aware of the statistics on the at least a hundredfold effort required to make changes to a program after the system has
become operational compared to making changes during development (13).

Also we are aware of the ratio of systems development effort to system
maintenance effort. At the Computer Systems Command the maintenance
effort is between 75 and 80 percent of the total life cycle cost.

Of course, maintenance does not mean simply to correct errors. The
vast majority of the work is concerned with putting the system in a
condition that the user "really wanted" in the first place.

Here we might note the lack of basic knowledge in our technology. While we are all horrified by the high cost of patching an error in delivered code, let me point out that one way to lower the per-error cost is to deliver even faultier code in the first place. Then the fixing would be averaged over a larger number of errors thus possibly improving the cosmetics of the statistic. Conversely consider a utopian code performing as required after a single error was corrected halfway through the life cycle. Obviously the software had to be maintained with some resources devoted to studying, reading, testing the code used by the "customers". The entire cost of this activity would have to be charged against a single error.

The entire field of software engineering suffers from such lack of basic data, of basic insight. We may have many measures, but do not know the invariants of the field: we are still waiting for the first orderly analysis which will define the analogs of the Mach, Froude and Weber numbers which moved fluid dynamics from an art to a science.

Returning to the discussion of user requirements and our need to aid users in formulating their needs, it seems obvious that we could profitably concentrate our attention on developing concepts, tools and techniques which enhance the user's ability to specify his needs early, but in such a way that recognizes his lack of full knowledge and assists him in redefining and refining those requirements which he stated initially. This redefinition would be based on an early assessment of the likely effects of meeting the requirements as originally specified. Such an approach would greatly reduce costly

systems maintenance and reduce the time required to have operationally effective systems.

Robert A. Frosch, past Assitant Secretary of the Navy for Research and Development pointed out that "anyone who has ever done a development or a design as opposed to setting up a management system for doing so, is well aware of the fact that the real world proceeds by a kind of a feedback process which looks more like a helix than a line". Hence we can state Problem Number Two as the lack of a meaningful feedback to the user in a timely enough fashion to allow him to monitor our understanding of his problem.

Thus we need to re-orient our thinking in two ways:

- 1. to recognize that the initial user requirements will typically not be true reflections of what the user actually needs and that his final requirements will come only after many changes will be made. Just the recognition of this fact would be a major improvement in our business.
- 2. To build a means by which we speed up the process for inexpensively "programing" the initial user requirements and furnishing the results in such a way as to facilitate making further refinements. This process would be repeated many times.

It sounds like heresy, but it may be more effective for the user to take far less time to define his "requirements" initially than he does now.

These "half-specified" requirements would then be given to the design team early to be used in a first cut at a product. The output from the first cut would be a familiarization with the requirements. Gaps in the

requirements would be filled in by the user on a demand basis.

In support of this way of system design, we need a conceptual change in the way we do business and much additional research and development leading to two classes of tools. The change in concept, which must be accepted by both the designer and user communities, would be explicit recognition of the essentiallity of the designer becoming more intimately familiar with the functional areas in which he is working and of a change in the way reqirements are developed. The familiarization would have to come from a direct interchange between the designer and the user as well as through formal training. The change in preparation of the requirements would involve the user preparing his initial requirements in a gross form, furnishing these to the system designers, who in turn would work directly with the user in putting these in a form from which the designer could prepare a simulation. Thus the first class of tools which must be developed are simulators. Simulations were used in the past for many purposes, but for the most part they were used for providing information on probable system runtimes, queue times and lengths, peak loading, and the like. We have not exploited the capabilities of simulators to produce answers to many of the basic questions that system users ought to be asking.

For example in an inventory control system obviously we need certain basic inputs: requests for supplies, receipt information, inventory data, and data on cancellations. The user, typically operating in an area of less than optimum information, will probably specify many other inputs which the system generates as an output at some earlier

time. For example there might be specifications on back-ordering information, substitutions, and the like. When the user specifies such data in the requirements statement he has no way of knowing the volumes of output and interactions of these transactions with one another. The results today are often the functional systems which generate paper so profusely and choke themselves on their own files and reports.

If the simulation tools are well developed and understood, the production of the simulations would not have to be terribly expensive. Based on the results of the early simulations the user would be in a position to make impact analyses in terms of cost, time, work-load and assess the impact on all the system's users. In the truest sense users include everyone who comes in contact with the system either through inputting to it, receiving outputs from it, or interacting through intermediate transactions. The simulation and modeling envisioned here would permit the assessment of the system on all its users.

After the simulations have been evaluated by the user and the requirements have been re-defined and purified through a process of iteration, the next step would be the production of a "breadboard" of the key modules of the system. The term "software breadboard" comes from the concept of hardware breadboards. One of the objections to the analogy between these two breadboards has been the tacit understanding that once a software breadboard is produced it is the final deliverable product. Unlike the equipment breadboards, which are used to make decisions regarding design choices and future production, we are accustomed to thinking of software as being fully usable once coded.

In the context used here the software breadboard is a product developed with little or no thought to programming efficiency, to computer run-time, or cost effectiveness of the breadboard itself. Furthermore, software bread-boards would be produced for only key modules of a system. For example, in an inventory control system, bread-boards might be produced only for the edit modules, the daily cycle, and report output modules - not for the myriad of other modules involved. The bread-board would not be a final specification. but would be used by the designer and the user to certify to the validity of a processing concept and as the basis for producing the final design specification. The bread-board would be "throw away" coding in all respects. This code is similar to an engineering hardware prototype in that it is used to demonstrate the capabilities of the design, is used to furnish data for iteration of changes, and is the basis for the user's acceptance of the work. It is also similar to hardware prototypes in that no cost has been spared in making it functional, but will be changed for release to the customer. I foresee a large simulation facility producing such prototypes, which would then be turned over to a production line for final "assembly" using affordable components.

Can we establish such a facility? What are the required ingredients? I see the need for a new breed of software artists, equipped with more scientific knowledge, who probably will have to be the "chief of chiefs" in respect to the programming team ideas. He must be innovative, must have a vast support facility, both in hardware, but especially in software, and must be allowed to operate unfettered.

He would be given much leeway in the selection of software items and design tools. He would test for accurate output, but would not bother testing for all possible outcomes in a given programming leg. He would write in a high order macro-language and would not concern himself with compiler effectiveness. The finished product would be a bread-board on which the user would either place his stamp of approval or request changes as appropriate. After the changes have been made and the bread-board approved, it, along with necessary documentation, would be given to a final design and programming group to be used for preparing the final design specifications. No repair, no modification, no "build-upon" the breadboard would occur. The breadboard would be thrown-away after it is no longer required.

The idea of the breadboard will require new classes of tools.

But the nucleus in the beginning of most of them exist now. These tools build on the demonstrated effectiveness and the essentiality of the structured design and programming technology.

The process outlined above for the production of the software breadboard would, of course, not be applied to production (i.e., deliverable) software which must be placed in maintenance. Completed software must conform to standards which are uniformly enforced in the designer community. An effort amounting to 15% of the programing just for establishing standards, enforcing them and verifying the code is not excessive. Independent third-party testing is considered essential for auditing code for standards compliance. It is these quality assurers who would have the final word on what code could be delivered.

What I have discussed so far are my views on developing software in the face of realistic user requirements. However, we all have the problem of "maintaining" already coded software systems. Maintenance consists mostly of changing the program to have it continue to meet changes in the user requirements. With already-running code we have the problem of identifying parts of the code which may have to be altered in response to a new user requirement, or, rarely, in response to a newly-discovered logic error, and in making the indicated changes without introducing errors elsewhere in the program.

The first need is to be able to read the existing code. While the training of the programmer stresses design of code and code writing, in fact most of the maintenance programer's time is spent in reading code. Since 75% of our effort may be spent in program maintenance, facility in code reading may be more important than a capability for code production. Here I borrow from Harland Mills: The training we give our programers stresses the wrong skills too early. An analogy from English literature teachings may be appropriate: Only after the student has achieved a measure of mastery of producing critiques of someone else's work is he allowed to produce his own creative prose. Harland Mills is proposing to teach programers to read code first, only later let him produce code.

To help the reading of code, it should be put into a standard structural form. An automated flow analysis and pattern recognition can help in producing a structured programing form as was shown by the restructuring engine concept. While there is no doubt that the resulting code, produced at about 20% of the cost of brand-new code,

is easier to read and may be cheaper to maintain, this represents but a first step in what should be the total maintenance activity.

Most of our large business codes are concerned with the maintenance, updating and cataloguing of very large files and data bases. Some of the more efficient methods and algorithms for searching, sorting and updating are but three to five years old. Most of the running codes are older, seven to ten years not being too atypical. Hence the more efficient algorithms are not in the codes being maintained.

While machine efficiency is not a major concern with most computer operations, many of us are faced with running the same code on say one hundred machines. In this case any small savings will be greatly multiplied; but even in cases where a given application may tend to saturate the computer system, changes in algorithms can result in substantial improvements.

You may have noted that nowhere did I refer to the kind of computer we would use. My concern is with meeting the user requirements first; part of the extended simulation I talked about earlier could result in defining a computing structure for the system. To be sure, today we think of huge machines for large systems. However, the hardware cost-trends seem to indicate that it is not unrealistic to start thinking about microprocessors (or small single-board computers) that have the processing power of an IBM 360/30, a machine quite adequate for many business applications.

In our work we software developers must continually realize that ultimately we serve the user; and in meeting our customer's requirements we must ourselves introduce the best applicable tools. While

we must know how to use all the newest tools, we must also know when not to use them. I suggest we must carry the latest and biggest sticks, but must tread very softly.

That our software costs are killing us, is no secret. Most likely a relief from these costs will come from a systematic introduction of more computer science into the art of computer programing. Without question it is necessary to be creative and artful in the exercise of our science, but we are getting into a mode where the bulk of the programers will have to attend to the day-to-day maintenance of codes, at least during part of their working life. We must therefore devote more attention to producing early verifiable and reliable code for satisfying users and more maintainable code for easing our own work.

My talk this evening touched on a few of the problems and visions for their solutions that have surfaced in the last year. Also I had the opportunity to get to know many of the leading thinkers in the software community, to help set a course for long-range solutions and to dream about the near-future of our business. Software engineering shows an exciting future. I foresee effective solutions to our current dilemmas, reachable by orderly work, by further exchange of ideas, for the germ of our long-term solutions appear to exist already. Symposia like this one mark orderly progress in our quest and point to further fertile developments.

On behalf of General Hancock, the Army Computer Systems Command and ISRAD I wish you a continued successful symposium.

#### References

- J. L. Hancock, Presentation at the Capitol Area Chapter of Society for Management Information Systems, Washington, D.C., 31 May 1977; to be published by Journal of SMIS.
- 2. Proceedings of the First Annual Software Symposium, U.S. Army Integrated Software Research and Development Program, Computer Systems Command, Fort Belvoir, VA, 23-24 May 1977.
- L. H. Putnam, "A General Solution to the Software Sizing and Estimating Problem", Life Cycle Management Conference, American Institute of Industrial Engineers, Washington, D.C., 8 February 1977.
- 4. L. H. Putnam, "A Macro-Estimating Methodology for Software Development", COMPCON '76, Thirteenth IEEE Computer Society International Conference, September 1976, pp. 138-145.
- Final Report on the Special Quality Assurance Review of the VIC Project, U.S. Army Computer Systems Command, Fort Belvoir, VA, May 1977.
- General Functional System Requirements, Direct Support Unit Standard Supply System, US Army Logistics Center, Fort Lee, VA, May 1975.
- Detailed Functional System Requirements, Direct Support Unit Standard Supply System, US Army Logistics Center, Fort Lee, VA, July 1976.
- 8. R. Mercier and J. Severin, "Conversion of a Logistics Supply System to a Minicomputer", Proceedings COMPSAC, Chicago, Ill., November 1977.
- Final Report of the MARDIS Multi-ADPE Conversion, US Army Computer Systems Command, Fort Belvoir, VA, August 1977.
- A. LeClair, "Transporting a Software System Between Alien Large Computers", Proceedings COMPSAC, Chicago, Ill, November 1977.
- 11. H. K. Desai, "Software Portability Study", Final Report on Contract DAHC26-76-D-1004, US Army Computer Systems Command, Fort Belvoir, VA, April 1977.
- 12. Research and Development Program, US Army Computer Systems Command, Fort Belvoir, VA.
- 13. B. W. Boehm, "Software Engineering", Proc. IEEE Transactions on Computers, December 1976, pp 1226-1241.

v

PAPERS PRESENTED

## SOFTWARE MAINTENANCE AND LIFE CYCLE MANAGEMENT

#### Clair R. Miller

## Honeywell Information Systems

I recall the early days of software projects where the assumption was made that software was deficient the minute you produced it. In the Univac I days the constraint of a thousand words of memory was the controlling factor of the entire programming process. It was obvious then that software had a short life cycle because everybody wanted something better. Phases in the production cycle like design, implementation, testing, and maintenance were tilted toward the front end, that is, the designing of new systems.

For business-oriented applications today, it is estimated that each instruction in a computer program costs \$10 to \$15. These are the same figures that were used in Univac I days so, by this standard, productivity has not increased. General Slay, <sup>1</sup> in a recent article in <u>Air Force Magazine</u>, quotes the cost of writing an average FORTRAN instruction at \$25. He states that the Air Force spends anywhere from \$100 to \$300 to write a software line of instruction for complex, real-time digital systems! For the SAGE Air Defense System back in the fifties, the cost for a line of instruction was estimated at \$50. For this kind of programming, productivity may have actually decreased.

Manufacturers of computers are vitally interested in software productivity, management control, and resource allocation for software projects. If you look at the balance sheet of a computer manufacturer it is hard to find the assets devoted to software. The corporate level allocations deal with factories, shipments, and

1 Slay, Alton D., Lt. Gen. USAF, "An Air Force Avionics Policy," Air Force Magazine, July 1977.

revenues tied to hardware. However, the cost of developing and maintaining software is becoming so great that it is dictating strategy in developing new hardware products.

Computer manufacturers' software is the interface for the user and the hardware and must meet the test of the competitive marketplace. In terminal-oriented systems, it is the main part the customer sees. This is somewhat different from applications software produced for one particular purpose and where accounting procedures are not a part of the management control procedures.

One phase of software production, irrespective of whether it is sold or used inhouse, is maintenance of software. This is the last phase in the life cycle.

The resource allocation of a software project, as it is usually conceived, is depicted in Figure 1.



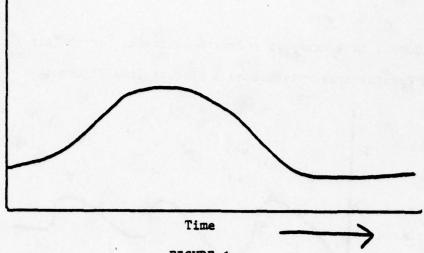


FIGURE 1

The low end of the curve is the maintenance phase which supposedly occurs after the large-systems design and programming effort. We usually plan and budget based on this curve. However, what usually happens is a line that looks like Figure 2.

Resource Allocation

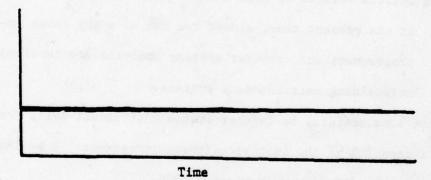
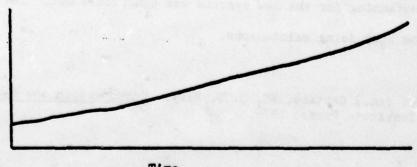


FIGURE 2

If inflation is considered, then the line looks like Figure 3.

Resource Allocation



Time

FIGURE 3

MITRE Corp., in a study of software contracts for the Air Force in the Command and Control area, postulated a picture like Figure 4.

Resource Allocation

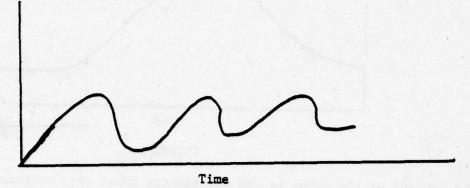


FIGURE 4

They found that large Command and Control projects were always done twice. I have a feeling if the analysts go back later, they will find they are in the third iteration.

In an article written by Rear Admiral Frank S. Haak, U. S. Navy, he stated:

At the present time, almost two out of every three in-house

programmers and computer systems analysts are involved in

maintaining existing data systems.

In the same article, he further stated that annual costs for software maintenance may well exceed 70% of the initial software investment. I believe Barry Boehm of TRW first hit the 70% figure in October 1976.

Recently, while I was investigating a large Air Force project where 1200 programmers are involved in replacing old systems, I asked what the programmers would do if all the programming for the new systems was contracted out. The answer was that they would be busy doing maintenance.

Prokop, Jan., Captain, SC, U. S. Navy. <u>Computers in the Navy.</u> Annapolis, Maryland: Naval Institute Press, 1976.

The evidence cited fits with that of a picture I believe exists for computer vendors; that is, the major portion of the cost of software over the life cycle is in maintenance. Cost data is scarce, but it is probably safe to say that 50% to 70% of total costs are involved.

If the major portion of the life cycle cost is maintenance, then there is more happening than just error correction. What is happening is that new functionality and instructions are being added. The software is evolving and the constraint of maintaining compatibility between the original design and the new additions causes the maintenance task to stretch out over a long period of time. Also, a dominant force now is change in architecture of systems. There is a need to update the hardware piecemeal which means you have old and new hardware within one overall system architecture. This also probably means old and new software.

Computer manufacturers are well aware of the problem of maintaining compatibility. When new machines are designed they carry forward the instruction set of old machines so that old programs will run on new machines. This is inefficient, not only for the old programs but for the new ones as well. The chances are more instructions than necessary will be executed in the new programs. Therefore, after a period of time, maintenance of programs becomes more and more complex. The old must be maintained as well as the new.

In studies and management guides devoted to software, only the development process is considered. There is an assumption that once it is developed, tested and declared operational, the system life cycle phases and major milestones are completed. Theoretically, the resources can be transferred to another project.

In reality, the resources are kept in place to maintain the software, and new people and new hardware are allocated to new endeavors.

Maintenance of software is a feedback process. The key to good feedback is

"Mass Use Overtime." The more the software is used, the better it gets if deficiencies

are fed back into the development group and corrections and modifications made.

Also, the longer it is used the less the probability that major deficiencies will

occur.

There are two principles that should help maintainability. In the initial design it must be clear that maintenance will have to deal with errors that are discovered based on mass use overtime and extensions and modifications of the software product in order to satisfy new requirements. Therefore, the programs must have readability. There has to be a deep understanding that comes from reading the programs over and over. This seems to be one of the main justifications for structured programming; that it simplifies the understanding and reading of programs. The structure has built in consideration for downstream support.

The second principle is modularity. The modules can be in almost any form.

Again, this is a fairly well recognized principle of structured programming. Breakpoints have always been there, but now more emphasis is placed on making it easier
for true modules to fit together.

These are principles that apply to the completely "new" program. They are sometimes hard to apply to a program that is a mixture of "old" and "new." This leads back to the basic assumption: programs have a life. At some point, anything that gets rid of the old is good. You can maintain for so long, and then the package should be scrapped. The logic that is used is that even if the new is bad it is really good because it gets rid of the old.

One of the difficulties in maintaining software is the fact that the process is not well defined. Equally undefined are the skills that are involved. People tend to specialize in certain kinds of programs such as compilers, data base systems, operating systems, etc., but nobody wants to specialize in program maintenance. Therefore, the tasks involved have not been defined, the skills involved have not been taught, and exact values to measure accomplishment is not available. Programming is essentially a mental discipline. The changes in this mental outlook which occur in going from a design phase to maintenance phase over the life cycle have been only partially defined.

There are a lot of behavioral science studies that lead to the conclusion that a group of 5 to 10 people, with a leader is the basic organizational structure for a development team. I could find no literature that applies in the context of a team for "program maintenance." Generally, it is understood that maintenance is one person out of a central location. Some kind of "bug report" is generated by users. These are controlled and go into a data base. The solution to a bug is not always programming. Sometimes, there is functional misuse, documentation faults, and hardware or operator faults. When a bug does involve programming, then the whole cycle, including test and evaluation, is triggered. Thus, each bug has a life cycle of its own. Also, bugs frequently not only affect the correct operation of the program in which the bug is situated, but also other programs or system operation and quite frequently system integrity. In a multiprogramming environment where the program executes in spurts, the beginnings of failure may have occurred a number of spurts before the one in which the crash eventually occurred.

Programmers can spend hours and days tracking down a single bug. Usually, this is the programmer you need the most for other work. If the pressure is heavy, everybody can get weary and confused.

There has to be a family of tools available. The operating system has to record recent events, there has to be breakpoint in the programs, and there has to be automatic test and diagnostic tools.

In DOD software management studies, the solution to less maintenance is perceived to be in better design and planning initially. The structured programming approach has this focus. It emphasizes the flow of information through the program and the transformation which occurs to that information. Top-down design emphasizes the entire system and the interrelationship among modules. Thus, the whole idea is to be able to go backward and see what is happening.

The technology of software development, which focuses on how you do it, rather than on the end product, is changing. It does not appear that it will reduce software production time or affect software productivity, but it does appear that it will produce more straightforward software and thus alleviate the maintenance problem. However, the need for close management control, planning, and attention to resource allocation will remain. As new distributed systems software becomes operational, emphasis on maintenance will become more and more important. Failures will be much more visible because systems are getting bigger, more money is involved, and distributed systems spread the mistakes over a larger geographic area. It will be necessary to have sophisticated maintenance people who have all the resources they need.

There has been progress over the last 25 years. It has been evolutionary rather than revolutionary. Things happen, not only because they are technically feasible but also because of the dynamics of the user-supplier relationship. Users don't want to have to redo everything. They want a little more than they have now. This little more is a maintenance function where software is involved. Such an incremental advance does not come out of universities or laboratories. With few exception, it comes from the interaction of users and suppliers and from small groups doing software maintenance. If maintenance is the way you progress, then resources allocated to this function should receive a higher priority.

#### REFERENCES

Proceedings of the Honeywell Software Productivity Symposium, April 26-28, 1977, Minneapolis.

Bremer, John W., "Hardware Technology in the Year 2001," Computer Magazine,
December 1976.

Spangle, C. W., "Remarks to National Conference on Computer Systems Productivity," June 27, 1977, Washington, D. C.

#### SOFTWARE MANAGEMENT STANDARDS

Dennis W. Fife
Institute for Computer Sciences and Technology
National Bureau of Standards
Washington, D. C. 20234

Today, providing computer software involves greater cost and risk than providing computer equipment, because hardware is mass produced by industry using proven technology, while software is still produced mostly by the craft of individual computer programmers and users. Over 70% of all software is developed by the users of computers, rather than the manufacturers of computer equipment or the producers of commercial software. Software is very expensive, and the development costs for new systems are growing to enormous levels. The Federal government is estimated to spend as much as \$5 billion annually on software, and its accumulated investment in current software probably exceeds \$25 billion. Software costs are projected to become 9 times as great as hardware costs by 1985.

This alarming trend has several causes. One of course is the declining cost of hardware, which gives impetus to new system concepts and expanding applications. Another is growing use of the "software approach" to hardware implementation (i.e. microprocessors rather than wired logic, provide control and intelligence). But the dominant reasons are the labor-intensive, often undisciplined character software production, and the lack of technical foundations and standard practices for developing reliable and valid software. Delivered software typically has errors at a rate of one in every 300 program statements. About 70% of software currently attributed to "maintenance", which costs are involves correcting latent errors, fixing original design flaws, and retrofitting software for changed requirements that were not anticipated in the original software development. Similarly, nearly 50% of current development costs typically are consumed in software testing -- to remove errors, to correct poor specifications, and to fill in missing requirements. Thus, our inability to control software costs mostly stems from our inability to control the design process and the quality of the end product.

Because we are failing so badly regarding the practical aspects of designing and producing software, "software engineering" has become recognized as the kind of discipline needed to evolve solutions. The software engineering movement, now about three years old, is distilling from past

experience the best ideas for managing software projects and exercising technical control over results. This paper discusses the content and approach for a potential standards effort that would assure "best practice", considering both the management aspects and the programming and design techniques for software projects [1].

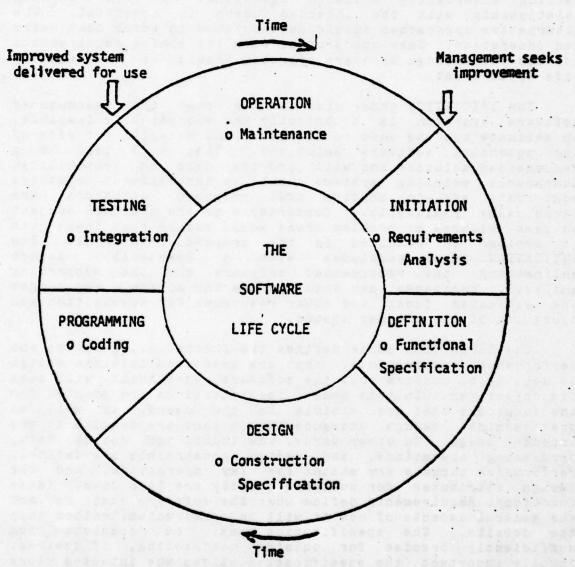
#### LIFE CYCLE MANAGEMENT

Software management consists of all the technical and management activities, decisions, and controls that are directly required to purchase, produce, or maintain software throughout the useful life of a computer system or service.

The basic framework for discussing software management is the life cycle viewpoint. The commonly recognized phases in the life cycle are shown in Figure 1, as described in software documentation standards [2]. The life cycle concept is not the only important notion, but it will always be dominant because it is a pragmatic view. After all, projects are paced by higher management actions, such as approvals, budgeting, and personnel assignments, which occur at distinct time points. Also, orderly progress on a complex undertaking requires a step-wise approach toward the final result, with each step an opportunity for evaluation and further preparation.

The INITIATION phase involves the assessment of an existing, inadequate system, in order to determine the feasibility of replacing it with an improved system. The four phases, DEFINITION through TESTING, are the development period, where a new or improved software product is being designed and implemented. The OPERATION phase starts with the operation of a new system by its intended users, and includes the subsequent maintenance and minor redesigns that user experience may dictate. Eventually, maintenance no longer suffices for needed improvements, and another development cycle would be started. The following summarizes the tasks included in each phase.

Figure 1. The Life Cycle of a Software Product



In the INITIATION phase, software management assures that the technical designers understand the purpose and scope of the envisioned system, and adequately perceive its required functions. The INITIATION phase begins when management recognizes that a need exists for a new or improved computer service, and that a study should be made to formulate a software solution. During this phase, software planners or system analysts identify the intended users and investigate their work processes. The planners describe the services needed from the envisioned software, and then conceive and outline alternative software solutions. A close working relationship with the intended users is essential. The alternative approaches should be described in terms that users can understand. User concurrence with the stated requirements and the recommended software approach should be a mandatory life cycle event.

The INITIATION phase also assures that the recommended software approach is technically and economically feasible. An estimate must be made of the costs and benefits for each of the potential software solutions. This will lead to a recommended solution and will provide data on feasibility. Comparable existing systems should be investigated to derive cost data and to confirm that proposed solutions technically realizable. Comparisons of the proposed project to past failures or problem areas would assure that these can be avoided or resolved in the proposed development. The INITIATION phase concludes with a Feasibility Report delineating the recommended software and the supporting analysis. Management may then approve the approach and budget the estimated funds and other resources for acquisition and operation of the planned system.

The DEFINITION phase defines the functional, quality, and performance requirements, that are needed to initiate design as well as to confirm that the software, when built, will meet its objectives. In this phase, specifications are created for the functions that are visible to the users, as well as predetermined design characteristics that are crucial to the intended usage. In other words, the input and output data, processing operations, and design constraints are defined. Performance targets are stated for key operations, and the design attributes for software quality are laid down. These Functional Requirements define what the software must do and the general aspects of how it will be constructed, rather than the details. The specification must be complete sufficiently precise for outside contracting, if desired. Equally important, the specification allows the intended users to confirm that the proposed system will meet the need and will have acceptable qualities of useability, generality, etc.

The DEFINITION phase also provides a Project Plan for managing the acquisition and installation of the system. The plan includes projections of the detailed costs. A delivery

schedule is laid out, including intermediate milestones and technical reviews for evaluating progress and insuring that the project will meet its goals. Working methods are defined for quality control of intermediate and final products. The plan should extend to the installation and initial operation of the system, and should include user training, document preparation and review, acceptance testing, and the continuing obligations of any contractors, vendors, or support organizations. The Project Plan is extremely important because it specifies how the effort will be managed, how problems will be resolved, and how the quality of the final system will be assured.

The DESIGN phase produces a thorough Design Specification for guiding a timely and problem-free implementation of the The DESIGN phase software. proposed begins Functional Requirements have been validated by intended users, and implementation of the system is approved. An early objective in this phase is to complete Design/Coding Standards reflect the quality requirements stated in DEFINITION phase, and which will be incumbent on every program produced. The DESIGN phase results in detailed specifications of the internal construction of the software, for use by programmers who will implement the design. This involves definition of the internal architecture of the software, performance analyses and selection of basic algorithms, delineation of individual programs or modules, specification of interactions between components of software, etc. Design Specifications and Design/Coding Standards are the means for resource management and quality control during PROGRAMMING.

In the PROGRAMMING phase, software management assures that high quality workmanship is applied in producing code meeting the Design Specifications. The PROGRAMMING phase often is merged with DESIGN and so may have no distinguishable starting point. PROGRAMMING involves final technical choices internal program design and the creation of programming for language statements or "code" that is executable on the computer and that implements the design. Software management here is the day-to-day supervision of programmers, guiding and reviewing their technical results. PROGRAMMING effort includes the testing by each programmer of his individual programs and the preparation of pertinent management reports and software documents. Test reports, inspection reports, and program documentation act as tangible quality controls over PROGRAMMING accomplishments.

In the TESTING phase, software management agsures that no significant errors or design discrepancies will impede usage of the system in the intended operational environment. TESTING proceeds according to a Test Plan prepared in the DESIGN phase, and focuses on the integration of individual programmer results in an overall working system. Noted

discrepancies and errors are documented for correction by the programming team. Gross shortcomings in the original design or functional specifications may be exposed; this should lead to a major project review in order to plan and organize additional definition and design work.

In OPERATION of a delivered system, software management assures that maintenance is well managed and controlled, with no less effectiveness than the original development. The OPERATION phase follows the delivery and installation of the software in the field, and involves periodic redesign and improvement. Software management procedures that are recommended during development can be streamlined to provide similar close control of the reduced effort committed to corrective design work. Thorough specifications, quality workmanship, effective testing, and management review remain highly important in directing technical effort to meet cost and schedule targets.

During the development period, the early tasks may be continued or repeated in order to remedy design defects and improve quality of the final system. Although the successive phases defined here should be followed whenever practical, the life cycle should be viewed with some flexibility, especially when a project involves unusual complexity or innovation. For example, it may be important to carry only a part of the system through to DESIGN and PROGRAMMING, in order to confirm feasibility before beginning DEFINITION for the entire system. So, all the software need not always be in the same phase of development.

The life cycle framework establishes a minimal set of standard milestones or goals for managing a software project, whether it is a completely new development or a major upgrading of an existing system. Figure 2 depicts the milestones that are of most concern to customers and upper management. In addition to these, the project manager must formulate intermediate milestones during the periods between these major events. Project milestones must be concrete, measurable indicators of progress. They must be directly observable by the project manager or customer, such as the receipt of a specified document, the successful operation of a given program, or the conclusion of a scheduled review meeting. They must be sufficiently frequent to serve as trouble alerts, and to give enough notice for action before serious repercussions result. Milestones should be formulated to permit correcting errors or omissions in specifications and software as early as possible.

Figure 2. Minimum Milestones for Life Cycle Management

NITIATION	DEFINITION	DESIGN	PROGRAMMING	TESTING	OPERATION

# Description of Milestones

- 1. Initiate project with plan for feasibility study.
- 2. Complete Feasibility Report with recommended software solution.
- 3. Complete Functional Requirements.
- 4. Complete Project Plan.
- Complete Test Plan; and Design/Coding Standards.
- 6. Complete Design Specification of all component programs.
- Complete unit testing; accept all program source code for integration testing.
- 8. Initiate fault reporting procedures.
- 9. Complete integration testing according to Test Plan.
- 10. Complete delivery and shakedown.

Each milestone in Figure 2 should be an occasion for intensive review that covers both technical and management issues. Participants would include higher management and customer representatives, but of course the key participant is the project manager. At each review, the project manager would present the current project status, giving expended versus allocated resources, any resource problems or technical concerns, and pertinent details of his planning to accomplish subsequent milestones. The deliverable product under review would be outlined briefly before commencing detailed comments and analysis. The reviewers should include technically knowledgeable people who are able to vouch for the adequacy of the product. Any discrepancies or needed improvements should be documented in a report, and accepted by the project manager for immediate rework and resolution. The milestone should not be considered complete until the modifications have been made or else follow-on milestones have been approved as changes to the Project Plan.

The process of periodic reviews is aimed at validating specifications and software results for responsiveness to the customer's need and consistency with the project resources. Each review presents management with the need to reassess the productivity of the project team and the resources needed for meeting the remaining project goals.

Continuing management involves both resource managment and technical control. If problems are evident in meeting cost and schedule objectives, management may seek additional resources or make better use of available resources. Or, management can decide to reduce the product quality, and act to deliver a poorer or lesser system within the existing cost and schedule targets. The latter course often may be taken, because higher management and customers usually cannot perceive this happening, and it postpones the potential day of reckoning.

The underlying source of software problems has been stated in various ways. Some authorities call it overambition, that is, attempting to build systems that are beyond our experience. Poor productivity is also claimed as the chief culprit. Both of these factors are certainly troublesome, but the more fundamental problem is probably poor quality control. To build systems on time and within cost, we must understand the relationship between project cost and resulting product quality, as well as the process of controlling quality as development or system operation proceeds. Yet, there are no commonly understood definitions of quality factors for software, and there is little reason to believe that the proven practices of quality software production are being widely adopted. A major goal in life cycle management should be to establish appropriate standards of practice that will provide quality assurance as a routine activity.

# THE PURPOSE AND NATURE OF STANDARDS

It is customary in computer standards to think in terms of a product standard, that is, a specification for some hardware or software item that may be purchased or built. In the general field of standards, however, considerable effort is spent in developing standards of practice, that is, methods and criteria for accomplishing tasks or analyses that are necessary to the engineering or construction of desired products. The only software standards that are close to being standards of practice are the few on documentation, which are mostly concerned with superficial mechanics, such as the layout of coding forms. Clearly, this is a poor situation, if you accept the idea that software design and development is not a routine task that is quickly learned and well performed by any person of average intelligence.

Standards of practice for software management and quality assurance could accomplish a number of important purposes. First, software managers need standards that would form an authoritative basis for estimating the minimum resources and schedule required for a proposed software project. In other words, standards could help resolve feasibility questions, and would prevent ill-advised customer demands for highest quality even though funds and schedules were being reduced below the feasible level.

Second, standards of practice would guide the new software manager who enters on this responsibility from another field, without adequate prior experience. Despite advances in professional education, many people still seem to be thrust into decision-making roles on projects, having only a general engineering or technical background, or routine programming experience.

Third, standards are necessary to instill an engineering approach and modern techniques into all projects. The rapid evolution of software technology, in less than a generation, presents us with too many people who have limited skills or outdated work approaches that cannot meet the quality requirements for modern complex systems.

On the other hand, standards of practice cannot be a substitute for adequate and continuing training. Standards should prescribe basic methods or the essential elements of techniques that are recommended, in a way that achieves simplicity of use and ready understanding for the technically oriented. Thorough understanding and fluent use would have to come through supplemental training aimed at establishing consistent use of the standard. The standard serves then as a control and a guidepost for everyday practice.

# THE QUALITIES OF SUPERIOR SOFTWARE

Software management may be used to control quality at an acceptably high level while conserving the time and resources needed for a project, or it may be used to maximize quality insofar as possible with the resources and time available. Quality in software is a complex issue. Nevertheless, there are general properties that are characteristic of high quality software:

CORRECTNESS--Programs perform exactly and correctly all the functions defined in the specifications, if available, or else in the documentation. This means that incorrect documentation is as serious as an incorrect program. Correctness is an ideal quality, that is rarely determinable, so a more practical quality is RELIABILITY.

RELIABILITY--Programs perform without significant detectable errors all the functions defined in the specifications or the documentation. High reliability indicates that programs are relatively trouble free in performing what they are claimed to do. But an equally important question is whether the functions and performance are adequate and suitable to a needed purpose. The latter quality is called VALIDITY.

VALIDITY--Programs provide the performance, all functions, and appropriate interfaces to other software components, that are sufficient for beneficial application in the intended user environment. This means the software, without additional programming or manual intervention, has the capabilities to do the job it is supposed to do. Validity is a quality of specifications as well as computer programs. Examples of an invalid program would be an interactive editor that had no online function for retrieving stored text for inspection, or a FORTRAN language compiler that had no DO loop implementation. Validity involves judgement of user requirements, and may change if the intended application or purpose is altered. Because poor reliability may render a needed function useless, reliability is necessary to validity.

RESILIENCE (or ROBUSTNESS) -- Programs continue to perform in a reasonable way despite violations of the assumed input and usage conventions. Input of unacceptable data, or an inconsistent command, should never cause a result that is astonishing and detrimental to the user--such as the deletion of any valid results obtained previously. Programs should include routine checks and recovery possibilities that are "forgiving" of common user and data errors. Resilience is related to the broader quality of USEABILITY.

USEABILITY--Programs have functions and usage techniques that are natural and convenient for people, showing good

consideration of human factors and limitations. For example, useable programs have few arbitrary codes for data in input or output, have consistent conventions in different operating modes, and provide thorough diagnostic messages for errors or violations of use.

CLARITY--The functions and operation of the programs are easily understood from the user manual, and the program design and structure are readily apparent from the listing of program statements. This means that documentation must be well written, but also that the program is carefully designed, with meaningful choices of variable names, use of known algorithms, frequent and effective comments in the program to describe its operation, and a modular structure that isolates separate functions for examination.

MAINTAINABILITY--Programs are well documented by manuals and internal comments, and so well structured that another programmer could easily repair defects or make minor improvements. Clarity is essential for maintainability. But maintainability also implies a wide variety of good design attributes, such as program functions that help to diagnose potential problems, e.g., periodic reports of status or control totals; or, general techniques that readily support changes, e.g., the isolation of constants, report titles, and other static data as symbolic items.

MODIFIABILITY--Program functions that might require major change are well documented and isolated in distinct modules. Maintainability is essential to modifiability. But modifiability means that a concerted effort was made to anticipate major changes, and to plan the software design so that they could be made easily.

GENERALITY--Programs perform their functions over a wide range of input values and usage modes. Programs are not limited to special cases or ranges of values, when the functions are commonly or reasonably extendable to a more general case.

PORTABILITY--Programs are easily installed on another computer or under another operating system program. Standard programming language is used, and hardware or other software dependent features are isolated for easy change.

TESTABILITY--Programs are simply structured and use general algorithms, to facilitate step-by-step testing of all capabilities.

EFFICIENCY/ECONOMY--Programs have high performance algorithms and conservatively use computer resources, such as main storage, so that the cost of program operation is low.

#### DESIGN STANDARDS

The above quality definitions should be prominent in the mind of every customer and software planner during the DEFINITION and DESIGN phases when specifications and design standards are being developed. Although the above are general statements, they are a starting point for evolving a set of uniform design standards that would serve as a foundation for meeting quality requirements. An approach is to refine the definitions, separating different aspects of each criterion, and then proceed to define specific features that would represent the achievement of each criterion when implemented in every program. Of course, there also may be special requirements that only pertain to one system function or program. For example, maintainability has aspects--capability to diagnose errors or problems, and capability to change programs. A general software feature that supports maintainability in regard to changing programs, is the use of only standard, high level programming language. special requirement that may be needed economy/efficiency might, for instance, be to implement some particular table search or selection algorithm in assembly language, rather than the high level language used for other modules.

The result of pursuing quality requirements in this way will be a design standard in the form of a checklist that can be used in team reviews or inspections, and in many cases, in automated code auditing. Although the need for such a design checklist seems obvious and long-standing, one hasn't been published yet. Programming textbooks sometimes have suggestions of this type scattered among the example programming exercises. The problems of software quality only recently have spawned a few books and papers that collect such "design wisdom" for programmers [6-9].

Program design standards must be sufficiently precise to be objectively measured on an individual basis. That is, satisfaction of each standard must be either automatically determinable by a program analyzer, directly observable by inspecting the code, or demonstrable or testable by running the program. This is the only way that a basis can be established for designing in quality and for measuring quality during development and at delivery.

# QUALITY CONTROL STANDARDS

A variety of quality control techniques have been practiced in software development, with some benefit. They are not consistently used. Sometimes there is an adequate reason for not using one or another, because the cost in

personnel time and effort is greater than the expected benefit. But, considerable effort is being made at this moment on guidebooks or standards to formalize the practice of some techniques in many organizations [1-5]. In addition to these references, NASA is preparing software management guidelines, and the IEEE Computer Society Technical Committee on Software Engineering is working to draft a standard for industry. Figure 3 is a concise listing of many of the accepted quality controls. It is clear that most of them are not automated or purely technical methods, but rather are judgemental methods relying on documentation.

Narrative documentation, rather than program statements or other symbolic codes, is the primary basis for quality control of software at this time, for several reasons:

- 1) Almost any degree of quality is possible for software; what degree can be afforded must be agreed through the project documentation.
- 2) No better means is now accepted for specifications than narrative documents.
- 3) Software is a "black box" to users, and documentation is needed to test and exercise it.

  Because of this, quality control standards to a large extent must address the format, content, and means of preparing and updating documents. Computer software for text editing, filing, and document formatting are important tools for quality control.

#### STANDARD SOFTWARE PRODUCTION TOOLS

Software production urgently needs more use of automated tools and less use of judgemental methods for quality control and implementation of program code. As shown in a recent analysis of projects [10], the factors with greatest impact on productivity are those involving project complexity, personnel experience, and customer interaction, where there is little possibility of technological help. So attention must turn to the routine technical aspects of software development that arise in the DESIGN, PROGRAMMING, and TESTING phases. programming languages and interactive programming support have proven that major improvements occur in programmer productivity and software reliability, through better software for programmer support. The trend to improve programmer tools, that started with these advances over a decade ago, needs to be reinforced. The evolution of structured programming has pointed out the value of program librarian and source code control systems, as well innovations in programming language features. Source code analyzers, even though often used today, need further

advancement toward a capability of auditing for conformance to design and programming standards. Basic research is needed on software testing aids of various types, such as automated test data generation, test evaluation and control, and fault diagnosis, particularly to evaluate their benefits in detecting various types of program errors or design flaws.

Despite the fact that software aids to programming are nothing new, surveys [11-12] indicate that a good repertoire of tools is seldom available or consistently used. A standardization effort is needed to stimulate availability of uniform, low-cost tools. Since the design of a specific tool may be highly dependent on the language of target programs or on the host operating system software, the appropriate approach would probably concentrate on functional specification of generic types of tools and their application, rather than a product standard for one universal tool of each kind.

# Figure 3. A List of Quality Controls

PROJECT RECORD -- Official journal of plans and decisions.

DOCUMENTATION STANDARDS -- Rules on scope, content, and format.

STRUCTURED SPECIFICATIONS -- Format coordination for traceability from requirements through testing.

QUALITY REQUIREMENTS--Separate specification of quality design criteria.

UNIT FOLDERS--Standard programmer records of module status and design in progress.

FAULT REPORTS--Reports of observed software errors in operation.

SPECIFICATION CONTROL -- Control of proposed changes to specifications.

SOURCE CODE CONTROL -- Control of changes to delivered program modules.

TESTING STANDARDS -- Uniform criteria for all testing.

INDEPENDENT TEST PLANS--Independently prepared tests of integrated software.

CONFIGURATION MANAGEMENT -- Control of overall system status and releases.

TEAM REVIEWS -- Conferences to audit design and completed programs.

INSPECTIONS -- Independent audit of program subsystems.

PROJECT INFORMATION -- Computer information support to resource management.

PROGRAM ANALYZERS -- Software to audit source code.

PRODUCTION TOOLS -- Compiler, editor, debugger, etc.

STRUCTURED PROGRAMMING--Improved design and programming technique.

TOP-DOWN DEVELOPMENT--Building programs starting from the application-oriented level.

## CONCLUSIONS AND RESEARCH IMPLICATIONS

There is no evidence or reason to believe that software development cannot be managed through many of the commonly recognized approaches to project management and control. Of course, software terminology must be brought in, and there are special milestones to be defined, but much of the framework of system life cycle management is applicable and useful. The long overdue step is to insure that it is used by every organization in every project, despite the modest extra effort required. Many government agencies are initiating standards or guidelines as directives on future projects.

But, regulation in terms of management procedures or project organization is not enough. Standards of practice are needed that lay down the technical methods and criteria that should be followed. Establishing appropriate standards is a significant research effort. A great deal of technical wisdom must be assembled, evaluated, and codified in understandable English and mathematics. Standards must be scaled to project resources and the life cycle objectives of a product. This means that the evaluation effort may be a considerable challenge, to determine properly the applicability of prospective methods to different projects.

Near future breakthroughs in such an approach are likely to concentrate on design and programming tools that automate many of the routine tasks of software development. Quality control is the area needing work, particularly toward auditing and validation of programs, and the attainment of thorough testing.

The area least subject to routine improvement is one which will still benefit from the human skills freed up by improvements elsewhere. This means design, and particularly the creative process by which a skilled person turns unstructured functional requirements into a feasible software concept, that can be implemented with a given amount of time and money. Significant help here could still come from standards that present model system designs as cost and quality benchmarks, or that offer nominal resource estimation procedures to guide management decisions on project investments and quality requirements.

#### REFERENCES

[1] Fife, D. W., Computer Software Management: A Primer for Project Management and Quality Control, Special Publication 500-11, National Bureau of Standards, to be published April 1977.

- [2] National Bureau of Standards, Guidelines for Documentation of Computer Programs and Automated Data Systems, FIPS PUB 38, 15 February 1976.
- [3] U. S. Air Force, Electronic Systems Division, Software Acquisition Management Guidebooks, · ESD TR 75-85, An Air Force Guide for Monitoring and

Reporting Software Development Status, September 1975 (AD-A016488)

ESD TR 75-91, Software Acquisition Management Guidebook: Regulations, Specifications, and Standards, October 1975 (AD-A016401).

ESD TR 75-365, An Air Force Guide to Contracting for Software Acquisition, January 1976 (AD-A020444).

ESD TR 76-159, An Air Force Guide to Software Documentation Requirements, June 1976 (AD-A027051).

ESD TR 77-22, Software Acquisition Management Guidebook: Life Cycle Events, February 1977.

ESD TR 77-130, Software Acquisition Management Guidebook: Software Development and Maintenance Facilities, April 1977.

Others in preparation.

- [4] U. S. Air Force, Aeronautical Systems Division, Management Guide to Avionics Software Acquisition, ASD TR 76-11, June 1976 (4 vols.).
- [5] U. S. Army, Computer Systems Command, <u>Software Quality</u>
  <u>Assurance Program Requirements</u>, MIL-S-52779 (AD), 5 April 1974.
- [6] Kernighan, B. W. and Plauger, P. J., The Elements of Programming Style, McGraw Hill Book Company, New York, 1974.
- [7] Ledgard, H. F., Programming Proverbs, Hayden Book Company, Inc., Rochelle Park, New Jersey, 1975.
- [8] Van Tassel, D., <u>Program Style</u>, <u>Design</u>, <u>Efficiency</u>, <u>Debugging</u>, <u>and Testing</u>, <u>Prentice-Hall</u>, Inc., Englewood Cliffs, New Jersey, 1974.
- [9] Ledgard, H. F. and Cave, W. C., "Cobol under Control", Communications of the ACM, November 1976, 601-608.
- [10] Walston, C. E. and Felix, C. P., "A Method of Programming Measurement and Estimation", IBM Systems Journal, n. 1, 1977, 54-73.

[11] Deutsch, D. R., <u>Appraisal of Federal Government Cobol Standards and Software Management: Survey Results</u>, NBSIR 76-1000, National Bureau of Standards, Washington, D.C., June 1976.

[12] Lientz, B. P. et al., <u>Characteristics of Application Software Maintenance</u>, Information Systems Working Paper 4-77, Graduate School of Management, UCLA, Los Angeles, California, December 1976.

#### \* YTILANU SARATTON OF SOFTWARE SUALITY \*

9. W. Boehm J. R. Brown W. Lipow

TRM Systems and Energy Group

## Keywords

software engineering quality assurance software quality software measurement and avaluation quality metrics quality characteristics management by objectives software standards software reliability tasting

#### Abstract

The study reported in this paper establishes a conceptual framework and some key initial results in the analysis of the characteristics of software quality. Its main results and conclusions are:

- Explicit attention to characteristics of softwere quality can lead to significant savings in software life-cycle costs.
- The current software state-of-the-art imposes specific limitations on our ability to automatically and quantitatively evaluate the quality of software.
- A definitive hierarchy of well-defined, well-differentiated characteristics of software quality is developed. Its higher-level structure reflects the actual uses to which software quality evaluation would be put; its lower-level characteristics are closely correlated with actual software metric evaluations which can be performed.
- A large number of software quality-evaluation metrics have been defined, classified, and evaluated with respect to their potential benefits, quantifiability, and ease of automation.
- Particular software life-cycle activities have been identified which have significant leverage on software quality.

Most importantly, we believe that the study reported in this paper provides for the first time a clear, well-defined framework for assessing the often slippery issues associated with software quality, via the consistent and mutually supportive sets of definitions, distinctions, guidelines, and experiences cited. This framework is certainly not complete, but it has been brought to a point sufficient to serve as a viable basis for future refinements and extensions.

# I. Introduction

Why Evaluate Software Quality? Suppose you receive a software product which is delivered on time, within budget, and which correctly and efficiently performs all its specified functions. Does it follow that you will be happy with it? For several reasons, the answer may be "no." Here are some of the common problems you may find:

- 1. The software product may be hard to understand and difficult to modify. This leads to excessive costs in software maintenance, and these costs are not trivial. For example, a recent paper by Elshoff(1) indicates that 75 percent of General Motors' software effort is scent in software maintenance, and that GM is fairly typical of large industry software activities.
- 2. The software product may be difficult to use.

  or easy to misuse. A recent and report?

  identified over \$10.000,000 in unnecessary
  Government costs are to ADP problems: many of
  them were because the software was so easy to
  misuse.
- 3. The software ornduct may be unnecessarily machine-decendent. Or hard to integrate with other Organis. This problem is difficult enough now, but as machine types continue to proliferate, it will get worse and worse.

Major Software Quality Decision Points. There are a number of familiar situations in which it is cossible to exert a strong influence on software quality, and for which it is important to have a good understanding of the various characteristics of software quality. Here are a few:

- Preparing the quality specifications for a <u>software product</u>. Formulating what functions you need and now much performance ispeed, accuracy) you need are fairly straightforward. Indicating that you also need maintainability or understandability is important, but much more difficult to formulate in some testable fashion.
- Checking for compliance with quality specifications. This is essential if the quality specifications are to be meaningful. It can clearly be done with a large investment of good people, but this sort of checking is both expensive and hard on people's morale.

<sup>\*</sup>Previously published in the <u>Proceedings of the Second International Conference on Software Engineering</u>, San Francisco, October 1976.

- 3. Making corpor design tradeoffs between development costs and operational costs. This is especially important because tight development budgets or schedules cause projects to skimp on maintainability, portability, and usability.
- 4. Software package selection. Here again, many users need a relative assessment of how easily each package can be adapted to their installation's changing needs and narchere base.

The primary payoff of an increased capability to deal with software quality considerations would be an improvement in software maintenance cost-effectiveness. Too little of a quality (e.g., maintaineality) translates directly into too much cost (i.e., the cost of life-cycle maintenance such as correction of errors and response to new user requirements). In view of this simple truth, in is rether supprising that more serious and definitive work has not been done to data in the area of evaluating software quality.

Previous Studies. Development of methods for evaluating software quality appears to have first been attempted in an organized way by subey and Hardwick. (3) The method of Ref. I was to define code "attributes" and their "metrics," the former being a prose expression of the particular quality desired of the software, and the latter a mathematical function of parameters thought to relate to or define the attribute. Attributes such as: "A<sub>I</sub> — mathematical calculations are correctly performed;" or "A<sub>S</sub> — The program is intelligible; "ort "A<sub>S</sub> — The program is easy to modify, "were each-further analyzed to define less abstract, i.e., more concrete, attributes capable of being directly measured as to whether the attribute is present in software to some degree (on a scale of 0 to 100). Although a detailed breakdown of each major attribute was given in the reference, only a few metrics were defined and not particular application was mantioned.

A later study (4) performed by the authors included the formulation of metrics and their application in a controlled experiment to the computer programs (approximately 400 Fortran statements each) independently prepared to the same specification. In this study, only a limited number of attributes were considered, primarily those corresponding to attributes As and As of Ref. 3, mentioned previously.

In addition, there was a deliberate difference in quality emphasis in the two programming efforts: one was done by a "hotshot" programmer who was encouraged to maximize code efficiency, and one by a careful programmer who was encouraged to emphasize simplicity. The main results of the study were:

- Ten times as many errors were detected in the "efficient" program (over an identical series of 1000 test runs);
- The measures of program quality were significantly higher on the "simple" program; thus, they were good indicators of relative operational reliability, at least in this context.

Concurrently, other authors were recognizing the significance of characterizing and dealing explicitly with software quality attributes. Wulf(5) identified and provided concise definitions of seven important and reasonably non-overlapping attributes: maintain—ability/modifiability, robustness, clarity, performance, cost, portability, and human factors. Abernathy, et al(6) defined a number of characteristics of operating systems and analyzed some of the

tradeoffs between them. Weinberg<sup>(7)</sup> performed experiments in which several groups of programmers were given the same assignment but different success criteria (development speed, number of statements, amount of core used, output clarity, and program clarity). For each characteristic, the highest performance was achieved by the group given that characteristic as its success criterion.

An increasing number of people were recognizing the importance of software quality, and dealing implicitly with software quality attributes by addressing the establishment of "good programming practices.", he book on programming style by Kernighan and Plauger(3) is the best example of this work. Also, a series of reports by CIRAD(9,10) on software maintainability provide some source material and some items such as the following checklist of practices and features enhancing software maintainability: conceptual grouping, top-down programming, modularity, meaningfulness, uniformity, compactness, naturalness, transferability, comments, parentheses, and names. A report by Warren on Software Portability(II) provides an excellent summary of alternative approaches to portability—e.g., simulation, emulation, interpretation, constrapping, nighter-order language features—with primary emphasis on the portability of language processors.

Recently, a number of initiatives have recognized the importance of explicitly considering quality factors in software engineering. For example, four presentations in the recent ALAA/ACT/IEEE/DOC Software Management Conference address the subject: CeRoze's presentation 122 identifies 'software quality specifications and tradeoffs" as a high-oriently 200 initiative: Kossiakoff(13) identifies seven attributes of 'good' software specifications; Whitaker' identifies twelve explicit quality goals for new COC programming languages; and Light's presentation 122 on software quality assurance identifies five important measures of software quality.

Key Issues in Software Quality Evaluation. Some of the major issues in software quality evaluation are the following:

- L. Is it possible to establish definitions of the characteristics of software quality which are measurable and sufficiently non-overlapping to permit software quality evaluation? This will be discussed in Section II on "Characteristics of Quality Code."
- 2. How well can one measure overall software-quality or its individual characteristics? This will be discussed in Section III on "Measuring the Quality of Code."
- 3. How can information on software quality characteristics be used to improve the software life-cycle process? This will be discussed in Section IV.

# II. Characteristics of Quality Code

Code is the realization of the software requirements and the detailed software design. It is the production article that directly controls the operations of the user's system. This section of the paper addresses the problem of characterizing the quality of the code itself. This section, and the following section on measuring the quality of code, are based orimarily on a study performed on the subject by TRM for the National Bureau of Standards, [10] and or subsequent work in the area at TRM, including a Software Reliability Study performed for Rome Air Development Center.

Initial Study Objectives and Conclusions. The initial objectives of the Characteristics of Software Quality" study<sup>1-5</sup>/were to identify a set of characteristics of software quality and, for each characteristic to define one or more metrics such that:

- Given an arbitrary program, the metric provides a quantitative measure of the degree to which the program has the associated characteristic, and
- Overall software quality can be defined as some function of the values of the metrics.

Although "software" can have many components such as functional specifications, test plans, and operational manuals, this study concentrated on metrics which could be applied to Fortran source programs.

The initial conclusions of the study are summarized below. First, in software product development and avaluation, one is generally far more interested in where and how rather than now often the product is deficient. Thus, the most valuable automated tools for software quality analysis would generally be those which flagged deficiencies or anomalies in the program rather than just producing numbers. This has, of course, been true in the past for such items as compiler diagnostics; one would be justifiably irritated with a mere statement that "1.17 percent of your statements have unbalanced parentheses."

Second, we found that for virtually all the simple quantitative formulas, it was easy to find counterexamples which challenged their credibility as indicators of software quality. Some examples are given below.

- A metric was developed to calculate the average size of program modules as a measure of structuredness. However, suppose one has a software product with n 100-statement control routines and a library of m 5-statement computational routines, which would be considered well structured for any reasonable values of m and n. Then, if n = 2 and m = 98, the average module size is 6.9 statements, while if m = 10 and n = 10, the average module size is 52.2 statements.
- 2. A "robustness" metric was developed for the fraction of statements with potential singularities (divide, square root, logarithm, etc.) which were preceded by statements which tested and compensated for singularities. However, often the operation is in a context which makes the singularity immossible; a simple example is that of calculating the hypotenuse of a right triangle:

# Z = SQRT(X==2 + Y==2)

3. Same "self-descriptiveness" metrics were developed for the number of comment cards, the average length of comments, etc. However, it was fairly easy to recall programs with fewer and shorter comments which were much easier to understand than some with many extensive but poorly written comments.

Third, we concluded that the software field is still evolving too rapidly to establish matrics in some areas. In fact, doing so would tend to reinforce current practice, which may, not be good. For example, the use of data clusters and automatic type-checking would invalidate some reliability matrics based on checking for mixed-mode expressions, parameter range violations, etc.

Finally, we concluded that calculating and understanding the value of a single, overall metric for software quality may be more trouble than it is worth. The major problem is that many of the individual characteristics of quality are in conflict: added efficiency is often purchased at the price of portability, accuracy, understandability, and maintainability; added accuracy often conflicts with portability via dependence on word size; conciseness can conflict with legibility. Users generally find it difficult to quantify their preferences in such conflict situations. Another problem is that the metrics are generally incomplete measures of their associated characteristics. To summarize these considerations:

- The desirable qualities of a software product vary with the needs and priorities of the prospective user.
- There is, therefore, no single metric which can give a universally useful reting of software quality.
- At best, a prospective user could receive a useful rating by furnishing the quality rating system with a thorough set of checklists and priorities.
- Even so, since the metrics are not exhaustive, the resulting overall rating would be more suggestive than conclusive or prescriptive.
- Therefore, the best use for metrics at this point is as individual anomaly indicators, to be used as guides to software development, test planning, acquisition, and maintenance.

Identification and Classification of Quality Characteristics. Having reached the above conclusion, it was decided to develop a nierarchical set of characteristics and a set of anomaly-detecting metrics. Our plan and approach were as follows.

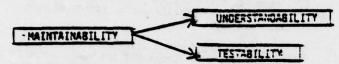
- Define a set of characteristics which are important for software, and reasonably exhaustive and non-overlapping.
- Develop candidate metrics for assessing the degree to which the software has the defined characteristic.
- Investigate the characteristics and associated metrics to determine their correlation with software quality, magnitude of potential benefits of using, quantifiability, and ease of automation.
- Evaluate each candidate metric with respect to the above criteria, and with respect to its interactions with other metrics: overlaps, dependencies, shortcomings, etc.
- Sased on these evaluations, refine the set of software characteristics into a set which is more mutually exclusive and exhaustive and supportive of software quality evaluation.
- Refine the candidate metrics and realign them in the context of the revised set of characteristics.

The following initial set of software characteristics were developed and defined as a first step: (1) (1) Understandability, (2) Completeness, (3) Conciseness, (4) Portability, (5) Consistency, (6) Maintainability, (7) Testability, (8) Usability, (9) Reliability, (10) Structuredness, (11) Efficiency, Definitions

of these characteristics are given in the Appendix.

As a second step, we their defined cardidate measurements of Fortran code (i.e., metrics, which would serve as useful indicators of the code's Understandability, Maintainability, etc. In doing so, we found that any measure of Understandability was also a measure of Maintainability—since any code maintanance requires that the maintainer understand the code. On the other hand, there were measures of Maintainability that had nothing to do with Understandability. For example, Testability features such as support of intermediate output and echo-checking of inputs are important to the retest function of software maintanance, but are unrelated to Understandability.

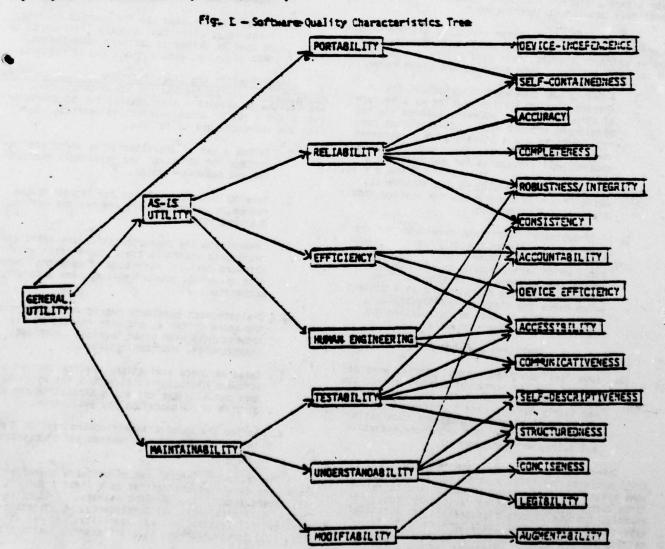
Thus, we began to find that the characteristics were related in a type of tree structure, e.g.,



in which the direction of the arrow represents a logical implication: if a program is Maintainable it must necessarily be Understandable and Testable; e.g., a high degree of Maintainability implies a high degree of Understandability and Testability. We also began to find that there was another level of more primitive concepts below the level of Understandability and Testability. For example, if a program is Understandable, it is also necessarily Structured, Consistent, and Concise (three of the original characteristics) and additionally Legible and Self-Descriptive (two additional characteristics not implied by the three above). We were thus generating some additional characteristics and finding that the entire set of characteristics could be represented in a tree structure, in which each of the more primitive characteristics was a necessary condition for some of the more general characteristics. (This result anticipated step. 5 of the plan outlined above.)

The resulting Software Quality Characteristics
Tree is shown in Fig. 1. Its higher-level structure
reflects the actual uses to which evaluation of softwere quality would be put. In general, when one is acquiring a software package, one is mainly concerned
with three questions:

- How-well (easily, reliably, efficiently) can: Luse it as-is?
- How easy is it to maintain (understand, modify, and retest)?
- . Car I still use it if I change my environment?



Thus, As-is Utility, Maintainability, and Portability are necessary (but not sufficient") conditions for General Utility. As-is Utility requires a program to be Reliable and adequately Efficient and Human-Engineered, but does not require the user to test the program, understand its internal workings, modify it, or try to use it elsewhere. Maintainability requires that the user be able to understand, modify, and test the program, and is aided by good Human-engineering, but does not depend on the program's current Reliability, Efficiency, or Portability (except to the extent the user's computer system is undergoing evolution).

The lower lev.! structure of the characteristics tree provides a set of primitive characteristics which are also strongly differentiated with respect to each other, and which combine into sets of recessary conditions for the intermediate-level characteristics. For example:

- A program which does not initialize its own storage is not completely Self-Contained and therefore is not completely Portable even though it may be completely Device-Independent.
- A program using formats such as 12A6 is not completely Device-Independent, and is therefore not completely Portable, even though it may be completely Self-Contained.
- A program which Device-Independent and Self-Contained but is not Accurate. Complete. Robust. Consistent. Accountable. Device-Efficient. Accessible. Communicative. Self-Descriptive. Structured. Concise. Legible. and Augmentable still satisfies the definition for Portability.

The primitive characteristics thus defined provide a much better foundation for defining quantitative metrics which can then be used to measure the relative possession of both the orimitive and the higher level characteristics. This can be done in terms which aid both in evaluating the utility of software products (at the high level) and in prescribing directions of needed improvements (at the primitive level). The definition and evaluation of such metrics is the subject of the next Section.

## III. Measuring the Quality of Code

Metrics. The term "metric" is defined as a measure of the extent or degree to which a product (here we are concentrating on code) possesses and exhibits a certain (quality) characteristic. As described in the previous section, we found that in fact, the development and refinement of both metrics and characteristics proceeded concurrently. Many metrics applicable to Fortran code were formulated and analyzed, leading in several iterations both to a refined set of metrics and to the generalized formulation of the hierarchical set of characteristics presented previously, we then had a basis for evaluating the usefulness of these entities, using the following criteria:

 Correlation with Software Quality. For each metric purportedly measuring a given 'orimitive" characteristic, did it in fact correlate with our notion of software quality? Here we mean roughly

There are subsets of applications in which additional characteristics are necessary: applications which require computer security, for example. A more detailed discussion of characteristics of secure systems can be found in Ref. 19.

by positive correlation that most computer programs with high scores for a given metric would also possess the associated primitive characteristic. Clearly, a more precise statistical definition could be stated, but the evaluation was quite subjective at this point, and the more precise measures of correlation would need to await extensive data collection and judgments on many diverse computer programs. The following scale was used to rate each metric:

- 4 Very high positive correlation; nearly all programs with a high metric score will possess the associated characteristic.
- AA High cositive correlation; a good majority (say 75-90%) of all programs with a high metric score will cossess the associated characteristic
- U Usually (say 50-75%) of all programs with a high metric score will possess the associated inaracteristic
- 3 Some programs with high metric scores will possess the associated characteristic.
- Potential Benefit of Metrics. Some metrics provide very important insights and decision inputs for both the developer and potential users of a software product; others provide information which is interesting, perhaps indicative of potential problems, but of no great loss if the metric does not have a high score even though highly correlated with its associated quality characteristic. The judgment as to its potential penefit is, of course, dependent on the uses for which the evaluator is assessing the product. The following scale of potential benefits was defined:
  - 5 Extremely important for metric to have a night score; major potential troubles other-
  - 1 important for metric to have a high score
  - 3 Fairly important for metric to have a high score
  - 2 Some incremental value for metric to have high score
  - 1 Slight incremental value for metric to have high score; no real loss otherwise.
- 3. Metric Quantifiability and Feasibility of Auto-mated Evaluation. While a metric may rate at the top for both correlation with quality and cotential benefit. it may be time-consuming or expensive to determine its numerical value. In fact, if to evaluate a metric requires an expert to read a program and make a judgment. the numerical value will generally provide much less insight than the understanding that the expert will pick up in the evaluation procass. Furthermore, metrics requiring excert inspectors are extravagantly excensive to use. Therefore, one would prefer for large programs an automated algorithm which examines the ororem and produces a metric value (and preferably also a list of local exceptions to possesston of the relevant characteristics). An intermediate capability which is often more fea-sible is an automated compliance checker, for which the user must provide a checklist of de-sired quality characteristics.

In the evaluation, a judgment was made as to which combination of methods of quantification would provide the most cost-effective rating for the metric, using the following set of options:

- AL car be done cost-effectively via an automated algorithm
- CC can be done cost-effectively via an automated compliance checker if given a checklist (Code Auditor is such a tool, described in Section IV)
- UE requires an untrained: inspector
- TT requires a trained inspector

- EI requires an expert inspector
- EX requires program to be executed.

Automating some evaluations, such as counting average module length or checking for the presence of certain kinds of self-descriptive material, can be done in a fairly easy and straightforward fashion. Automating other evaluations, such as scanning the program globally for repeated subexpressions and guaranteeing that the components in the subexpressions are not modified between repetitions, are possible but more difficult. Others, such as judging the descriptiveness of the self-descriptive material as well as its presence, are virtually impossible to automate. Thus,

Table I
EVALUATION OF QUALITY METRICS

Primitive Characteristic	s Definition of Metrics	Correlation with Quality	Potential Benefit	Quantifi- ability	Ease of Developing Automated Evaluation	Complete- ness of Automated Evaluation
Device- Independence DI-1	Are computations independent of com- puter word size for achievement of required precision or storage scheme?	A	<b>5</b>	AL + EX -TI	E	,
2-10	Have machine-dependent statements been flagged and commented (e.g., those computations which depend upon computer herdware capability for addressing half words, bytes, selected bit patterns, or those which employ extended source language features)?			AL		
Self- Containedness SC-1	Does the program contain a facility for initializing core storage prior to use?		5	Æ	E	P
2C-5	Does the program contain a facility for proper positioning of input/ output devices prior to use?	<b>A</b>	5	CE	•	
Accuracy AR-1	Are the numerical methods used by the program consistent with appli- cation requirements?	A	5	π		
AIT-2	Are the accuracies of program con- stants and tabular values consis- tent with application requirements?	A self of	5	AL + TE		To the ser
CP-1	Are all program inputs used within the program or their presence explained by a comment?	u	3	AL.	E	С
CI-S	Are there no "dummy" subprograms: referenced?	2	2	L	•	- 6
Robustness R-1	Does the program have the capabil- ity to assign default values to non-specified parameters?	A constant	5	AL + TI	ε	10 mg
2-4	Is input data checked for range errors?	W	5	AL + TI	ξ	
Consistency CS-1	Are all specifications of sets of global variables (i.e., those appearing in two or more subprograms) identical (e.g., labeled COMMON?	M	. •	<b>A.</b>	E	<b>Č</b>
C2-2	Is the type (e.g., real, integer, etc.) of a variable consistent for all uses?	77 5 70	5	AL.	•	and the

some automated tools could provide useful but only partial support to quality availation, leaving the remainder to be supplied by a numan reader. The following scales were used to rate each metric with respect to ease and completeness of automated avaluation:

# Ease of Developing Automated Evaluation

- E Easy to develop automated algorithm or compliance checker
- M Moderately difficult to develop automated algorithm or compliance checker
- 0 Difficult to develop automated algorithm or compliance checker

# Completeness of Automated Evaluation

- C Algorithm or checker provides total evaluation of metric
- P Algorithm or checker provides partial evaluation of metric
- I Algorithm or checker provides inconclusive

Evaluation of Metrics. The above criteria were applied to the candidate metrics developed in the study. Some examples are given in Table 1 (displayed on the previous page). Only a small fraction of the 151 candidate metrics (16/could be included here, but the ideas are amply illustrated. In Table 1 are snown, for each of six primitive characteristics, the evaluation of two of its associated metrics. An explanation of the ratings established for the first metric in the table is given below for clarity.

The metric (DI-1) was found to be highly correlated with Device-Independence (rating "A"), and to be extremely important with respect to Device-Independence (rating "5"). It was found that a communation of automated algorithm, execution, and a trained inspector would generally be most cost-effective for determining to what degree a software product possessed the characteristic (rating "AL + EX + TI"). An automated algorithm could check format statements for device-dependence, e.g., 1286, or F15.11 (more precision than most machines possess), and similarly flag extra-precise constants (e.g., PI = 3.14159265359). These checks would be easy to automate (rating "E"), but would only provide partial results (rating "P").

Evaluation of Metrics Versus Project Error Experience. The pest opportunity to evaluate the metrics and ratings exemplified above in Table 1 presented itself in the availability of an extensive data base of softwere error types and excerience in detecting and correcting them, compiled by TRW for the Air Force CCIP-d5 study. The initial segment of this data base is presented in Table 2 (displayed on the following page). It includes the classification of 224 software errors typed into 13 major categories:

- 1. Errors in preparation or processing of card input data
- Tape handling errors Disk handling errors
- Output processing errors
- Error message processing errors
- Software interface errors Hardware interface errors
- Data base interface errors User interface errors
- Computation errors

- 11. Indexing and substitution 12. Iterative procedures errors Indexing and subscripting errors
- 13. 31t manipulation errors

In the earlier study, all the errors of each type were analyzed to determine during which phase of the software development process they were typically (but not always) committed and where they were typically (but not always) found and corrected. The onases used in this analysis were:

- 1. Requirements Definition
- Casign
- Code and Debug
- Development Test
- 5. Validation
- Acceptance
- 7. Integration
- 3. Delivery

In Table 2, for each error type, an "O" is placed in the column corresponding to the shase in which that type of error typically originated, and an "F" is placed in the column corresponding to the phase in which that type of error was typically found and cor-rected. To evaluate the applicability of each metric to error detection and correction, an additional item was estimated for each error type: the phase in which that type of error would most likely be detected and corrected using that metric.

Example. Line 1 in the table indicates that the typical error of the type in which the "program expects a parameter in a different format than was given in the Program Requirement Specification, originated in the Design phase (at least for the software projects analyzed in the CCIP-85 study). That type of error was typically corrected during the acceptance testing phase. However, if metric CS-13 (an extension of metric-SD-1 covering header commentary) had been computed, this type of error would typically have seen caught and corrected in the design phase.

As is evident from this example, one result of the analysis was to identify several extensions of the previous metrics which would be effective in error detection and correction. For example, the CS-13 capability cited in the table implies the need for an automated tool which would scan standard software module needer blocks. It would eneck the consistency of their information with respect to assertions in the neader blocks about the nature of inputs and outputs, including:

- . data type and format
- number of inputs
- e order of inputs
- e units
- · acceptable ranges
- · associated storage locations
- · source (device or logical file or record)
- · access (read-only, restricted access)

The CS-13 entries in the table indicate that if coding of the module had been preceded by such a module de-scription with the assertions about its inputs and outputs, then an automated consistency checker could generally have caught the error before coding began.

Table 2. Evaluation of Error-Detecting Capabilities (Metrics) vs. Error Type (first 12 of 224 error types)

		1 7	3	4	5	5	7	8
Err	Saftware Phase	Requirement Dosign	Code	Devalopment Test	Validation	Acceptance	Integration	pelivary
rro	rs in Preparation or Processing of Card Input Data							
ı.	Program expects parameter in different format than is given in Program Requirement Specification.	CZ-13						
2.	Program does not expect or accept a required parameter.	CZ-13					F	
3.	Program expects parameters in a different order than that which is specified.	C2-13					F	
\$.	Program does not accept data through the entire range which is specified.	C2-13				F		
5.	Program expects parameter in units different from that which is specified.	C2-13				F		
6.	Nominal or default value utilized by program in the absence of specific input data is different from that which is specified.	α					F	
T.	Program accepts data outside of allowable range limits.	0						F
8.	Program will not accept all data within allowable range limits.	a						F
9.	Program overflows core tables with data that is within the allowed range.	a		CP-9				F
(T.	Program overflows allotted space in mass storage with data that is within the allowed range.	α		CP-9	1			F
11.	Program executes first test case properly but succeeding test cases fail.	Œ				F		
12_	Program expects parameter in a different location than specified.	Q-13				F		

0 = Error origin CS-N = Consistency-checking aid N applied at this phase would generally have de-F = Error found tected error.

CP-IT = Completeness-checking aid N applied at this phase would generally have detected error.

Of the 12 types of card processing errors shown in Table 2, the CS-I3 consistency checker would have caught 6. Overall, out of the 224 types of errors, this capability would generally have caught 18. The next most effective capability would perform checks on the consistency of the actual code with the module description produced during the design phase for capability CS-I3 above. (For example, for each output assertion, it would check if the veriable appeared on the left of an equals sign in the code, and perform a units check on the computation.) This capability would have caught 10 types of error, but not until the initial codescending phase.

In general, as is seen in Table 3, the Consistency metrics were the most effective aids to detecting software errors. Overall, they would have caught 34 of the 224 error types; their total phase gain (the sum of the number of phases that error detection was advanced by the metrics) amounted to 89, or an average gain of 2.5 phases per error type. The next most effective metrics were those for Robustness, followed by Self-Containedness and Communicativeness.

Table 3
ERROR CORRECTION EFFECTIVENESS OF METRICS

Metric/Primitive Characteristic	Corrected (No.)	Phase Gain (Total)
Conststency Robustness Self-Containednes Communicativeness Structuredness Self-Descriptiven Conciseness Accuracy Accuracy	9 2	29 17 28 18 2 4 4

The main message of Table 3 is that the early application of automated and semiautomated consistency. Robustness, and Self-Containedness oneckers leads to significant improvements in software error detection and correction. This is an important conclusion, but it should not be too surprising, since Consistency, Robustness, and Self-Containedness are three of the primitive characteristics associated with Reliability.

Another useful consistency check was to compare a metric's error-correction potential with the estimate of a metric's potential benefit in Table 1. Satisfactorily, virtually all of the significant error-detecting and correcting metrics had maximum potential penefit ratings of 5, and none which contributed to error correction had ratings less than 3.

Of course, this was just a partial evaluation, but since testing occupies such a great proportion of a total software effort, the above evaluation has been a most useful one for helping us to decide which metrics should have high priorities for development and use. We have subsequently developed some of these metric-checkers and used them with some success, and, in particular, the CS-13 metric was used as a basis for the recently developed Design Assertion Consistency Checker described in Ref. 20.

# IV. Using Quality Characteristics to Improve the Software Life-Cycle Process

The software life-cycle process begins with a system and software requirements determination phase, followed by successive phases for system design, detailed design, coding and testing, and culminating in an operations and maintenance phase. There are four major means we have found for using the quality characteristics discussed above to improve the life-cycle process. These are:

- Setting explicit software quality objectives and priorities;
- o Using software quality checklists:
- Establishing an explicit quality assurance activity;
- o Using quality-enhancing tools and techniques.

Explicit Software Quality Objectives and Priorties. The experiments reported in Refs. 4 and 7 showed that the degree of quality a person puts into a program correlates strongly with the software quality objectives and priorities he has been given. Thus, if a user wants portability and maintainability more than code efficiency, it is important to tell the developer this, preferably in a way which allows the user to determine to what extent these qualities are present in the final product.

Probably the best way to accomplish this to date is through the practice of <u>software quality benchmarking</u>. Benchmarking is generally just used to determine Device-Efficiency on a typical operational profile of user jobs, but it can be used similarly as an acceptance test or software package selection criterion for other qualities also. Thus, for maintainability, one constructs a representative operational profile of likely modifications to the software, and measures now efficiently and effectively these modifications are made by the actual software maintenance personnel.

Used as an acceptance test, software quality benchmarking provides an explicit set of quality objectives for all participants throughout the various

chases of the software development cycle. It has been used primarily to date in specifying nardware-software reliability and availability objectives (with particular success in the dell Labs' Electronic Switching System, for example), but has also been used successfully for other quality objectives.

Software quality benchmarking can and should be used also to evaluate alternative software products for procurement. In doing so, the level of effort expended in quality benchmarking should be proportional to the amount of use the product is expected to have, rather than to its price. More than we once, we have lost over \$10,300 in software development and operational costs because of incomplete quality benchmarking in procurement of \$1,000-\$2,000 software products.

Software Quality Checklists. The quality metrics summarized above, and presented in tetail in Ref. 15, can be used as the basis for a set of software quality checklists. These can be used to support reviews, walkthroughs, inspections, and other constructive independent assessments of a software development product. Again, to date, these have been used orimarily to support software reliability objectives. But can be used effectively for other software quality objectives. For example, Table 4 gives a portion of a shecklist for judging the Self-Descriptiveness of a computer program, an important characteristic in evaluating the long-term costs of understanding, testing, and maintaining the program.

PARTIAL CHECKLIST FOR JUDGING THE SELF-DESCRIPTIVENESS OF A SOFTWARE PRODUCT

A software product possesses self-descriptiveness to the extent that it contains enough information for a reader to determine or verify its objectives, assumutions, constraints, inputs, outputs, components, and revision status. Checklist:

- a. Does each program module contain a header block of commentary which describes (1) program name, (2) affective date. (3) accuracy requirement, (4) purpose, (5) limitations and restrictions, (6) modification history, (7) inputs and outputs, (8) method. (9) assumptions, (10) error recovery procedures for all foreseeable error exits that exist?
- b. Are decision points and subsequent branching alternatives adequately described?
- c. Are the functions of the modules as well as inputs/ outputs adequately defined to allow module testing?
- d. Are comments provided to support selection of specific input values to permit performance of specialized program testing?
- e. Is information provided to support assessment of the impact of a change in other portions of the program?
- f. Is information provided to support identification of program code which must be modified to effect a required change?
- g. Where there is module dependence, is it clearly specified by commentary, program documentation, or inherent program structure?
- h. Are variable names descriptive of the physical or functional property represented?
- f. Do uniquely recognizable functions contain adequate descriptive information (e.g., comments) so that the purpose of each is clear?
- j. Are adequate descriptions provided to allow correlation of variable names with the physical property or entity which they represent?

Quality Assurance Activity. We are finding it increasingly advantageous, from ooth product quality and cost-effectiveness standpoints, to have an explicit quality assurance activity on our software projects. The manager of this activity generally reports to the project manager and is purposely held from producing any of the deliverable product in order to provide an independent view of the project. Tasks included in this quality activity are tailored to the project and depend upon the size and scope of the project. This approach has proven effective in ensuring that the project is responsive to the quality requirements of the customer and the particular system application. The responsibilities of the quality assurance activity generally include:

- PTanning Preparation of a software quality
  assurance plan which interprets quality program requirements and assigns tasks, schedules
  and organizational responsibilities.
- Policy, Practice and Procedure Development —
  Preparation of standards manuals for all phases
  of software production, including requirements,
  design, coding, and test, tailored to specific
  project requirements. A key point here is attention to quality provisions early in the
  software life cycle.
- Software Quality Assurance Aids Development -Adaptation and development of manual and automated procedures for verifying compliance to: software functional and performance requirements and project quality standards.
- Audits Review of project procedures and documentation for compliance with software development plan standards, with follow-up and documentation of corrective actions.
- Test Surveillance Reporting of software probless, analysis of error causes and assurance of corrective action.
- Records Retention Retention of design and software problem reports, test cases, test data, logs verifying quality assurance reviews and other actions.
- Physical Media Control Inspection of disks, tapes, cards, and other program-retaining media: for verification at all times of physical transmittal or retention, and assurance that contents are not destroyed or altered by environment or mishandling.

To date, most of these responsibilities involve considerations of reliability and software product compliance to standards and the software requirements specification. However, the quality assurance activity can also be a useful focal point for assuring that the product possesses the other characteristics of software quality, such as Maintainability and Portability.

Quality-Enhancing Tools and Techniques. Practically every software tool and techniques-cross-reference generators, flow charters, configuration management procedures, software monitors—succorts some kind of quality-enhancement with respect to at least one quality characteristic. Here we concentrate on several tools and techniques having particularly nigh leverage for software quality enhancement, first those which apply to software requirements and design specifications and then those which apply to code.

Quality-Enhancing Tools and Techniques: Requirements and Design. During requirements and design phases, some assurance that desired quality characteristics are present can be obtained by using guidelines and detailed checklists. Here, of course, the primary objective is not to optain a numerical measure of the extent to which a quality characteristic is present. but to identify problems of varying levels of criticality with which we need to deal. A great deal of software quality leverage is gained by using machine-analyzable software specifications and automated aids to analyze them for Consistency, Completeness, etc., such as ISDOS(ZI) and SREP(22.22), provide for software requirements. These systems provide a significantly increased assurance of specification quality over manual methods, giving considerable leverage for limiting the cost of software errors during both testing and maintenance. This is because a majority of errors ariseowing to faulty expression of requirements and incom-plets design [20], and are much less expensive to correct during the early phases of production than duringsubsequent phases.

One of the biggest sources of software problems stems from ambiguity in the software requirements specifications. A number of different groups—designers, testers, trainers, users—must interpret and operate with the requirements independently. If their interpretations of the requirements are different, many development and operational problems will result.

One of the best counters to this problem is a review to make sure that the requirements are Testable.

For example, consider the pairs of specifications below.

#### Non-Testable

#### Testable

- I. Accuracy shall be I sufficient to support mission planning
- Z. System shell provide Z. real-time response to status queries:
- I\_ Position error shall be: ≤ 50° in the horizontal ≤ 20° in the vertical
  - System shall respond to:
    Type A queries in < Z
    sec
    Type & queries in < 10
    sec
    Type C queries in < 2
- 3. Terminate the simula- 3. tion at an appropriatesnift break

Terminate the simulation after 3 hours of simulated time.

It is clear that the specifications on the right are not only more Testable but also less amoignous and better suited as a baseline for designing, costing, documenting, operating, and maintaining the system.

There is one technique for explicitly analyzing such quality considerations during the requirements phase which has been reasonably successful on smail-to-medium projects. This is the Requirements-Properties Matrix: a matrix whose column consist of the individual functional requirements and whose rows consist of the major qualities (or properties) desired in the software product (or vice versa). The elements of the matrix consist of additional specifications which arise when the considers the quality implications of each requirement. For example, consider the third pair of reduirements above when treated in a Requirements-Properties Matrix as in Fig. 2. It is clear that the resulting specifications will lead to a higher quality software product. Some additional effort would be necessary to achieve the enhanced product, but if the program is a have a good deal of use and maintenance, the effort will pay off in reduced life-cycle costs.

COMPUTER SCIENCES CORP ARLINGTON VA F/G 9/2
SOFTWARE PHENOMENOLOGY - WORKING PAPERS OF THE SOFTWARE LIFE CY--ETC(U)
AUG 77 B ELKINS, L HUNT DAHC26-76-D-1006 AD-A053 014 NL UNCLASSIFIED 2 OF **8**ADA
053014 

Fig. 2 - Portion of a Requirements-Properties Matrix

Requirement Property	Terminate the simulation at an appropriate shift break	
Testability .	Terminate the simulation after 8. hours of simulated time	
Modifiability	Allow user to specify termination time as an in- put parameter, with a de- fault value of 8 hours	
Robustness	Provide an alternate termination condition in case the time criterion cannot be reached.	
	•	

Quality-Enhancing Tools and Techniques: Code.

As shown in the detailed characteristics tree of Fig.

1. code Structuredness is one of the necessary primitive characteristics for Testability, Understandability, or Modifiability; all of the latter are necessary for Maintainability. A set of allowable Fortran constructs for the basic control structures SEQUENCE.

IFTHENEISE, CASE, OOWHILE, and DOUNTILICAL were developed as a standard for Structuredness on a large real-time software project. An automated Fortran source code scanning program called STRUCT was developed and is regularly used to determine for each routine whether it is a properly nested combination of the allowable constructs; and when violations are recognized the code causing the violation is identified and a diagnostic issued.

The discipline invoked by this quality requirement on the particular project met with a certain amount of resistance and disgruntlement by programmers. Functional team leaders were somewhat dismayed at first since routines previously coded before the standard had been required needed to be redone to a great extent, which consequently strained labor cost budgets and made the original schedules difficult to meet. Subsequently, however, in a survey of programmers and their supervisors, most were of the opinion that maintenance costs would be reduced, in addition to expressing positive opinions on "quality of code" including consistency and understandability. The opinion was also expressed that had the standard been invoked in the first place, most of the development problems would have been avoided.

Subsequent evaluations of software errors observed in testing on the referred-to project have shown an extremely low rate, believed to be partly attributable to the application of the Structuredness standard, although the requirement to exercise all branches of a routine prior to turnover for integration testing was found to be much more influential in reducing the number of errors.

The structuring standard is simply one of over 30 other coding standards formulated for this software project, and automatically checked by a Fortran source code analysis tool called CODE AUDITOR. This tool determines whether these standards are violated, and shows the source code location of the violation in a diagnostic printout. Many of the primitive characteristics of Fig. 1 are measurable by CODE-AUDITOR. For example, Self-Descriptiveness is measured in part by

checking for the presence of standard module header commentary cards, and for commentary cards to explain transfers of control. Consistency is measured in part by checks for mixed-mode expressions, and compliance with standards for parameter passing.

In the future, some of these will be done more efficiently through standard language features for structured programming, data typing, etc. Yet there will remain a need for automatic post-scanning of code to assure compliance to local standards (e.g., naming conventions) and to check for partial indicators of potential quality problems.

Of course, as is evident from the ratings in Table 1, there are some evaluations of code (and design) quality which require trained humans to perform. Reference 25 reports some experiences on large software projects with emphasis on the relative merit of a variety of techniques involving both human inspectors and automated tools for controlling and measuring software quality. One particularly valuable technique in this regard is that of the design or code inspection (25) and its "cousin" technique, the structured walkthrough. In our analysis (17) of software errors found in the validation phase of one large project, we determined that 58 percent of them could nave been eliminated early by appropriate design inspection techniques. Fagan's results (20) on one fairly well-controlled project indicate that the extra effort involved in performing inspections paid off in a 23 percent reduction in operational errors.

#### V. Conclusions

Explicit attention to characteristics of software quality car lead to significant savings in software life-cycle costs. (Section I).

The current software state-of-the-art imposes specific limitations on our ability to automatically and quantitatively evaluate the quality of software (Section II).

A definitive hierarchy of well-defined, well-differentiated characteristics of software quality has been developed. Its higher-level structure reflects the actual uses to which software quality evaluation would be put; its lower-level characteristics are closely correlated with actual software metric evaluations which can be performed (Section II).

A large number of software quality-evaluation metrics have been defined, classified, and evaluated with respect to their potential benefits, quantifiability, and ease of automation (Section III).

Particular software life-cycle activities have been identified which have significant leverage on software quality (Section IV). These include:

- Setting explicit software quality objectives and priorities;
- · Performing software quality benchmarking;
- . Using software quality checklists;
- Establishing an explicit quality assurance activity;
- Using macrine-analyzable software specifications:
- . Ensuring testable software requirements;

- . Using a Requirements-Properties Matrix;
- Establishing standards, particularly for structured code;
- Using an automated Code Auditor for standards compliance checking;
- · Performing design and code inspections.

Most importantly, we believe that the study reported in this paper provides for the first time a clear, well-defined framework for assessing the often slippery issues associated with software quality, via the consistent and mutually supportive sets of definitions, distinctions, guidelines, and experiences cited. This framework is certainly not complete, but it has been brought to a point sufficient to support the evaluation of the relative cost-effectiveness of prospective code-analysis tools presented in this paper, and to serve as a viable basis for future refinements and extensions.

#### References

- Elshoff, J. L., "An Analysis of Some Commercial PL/I Programs," <u>IEEE Trans. Software Engineering</u>, June 1976, pp. 113-120.
- Improvements Needed in Managing Automated Decisionmaking by Computers Throughout the Federal Government. U.S. General Accounting Office, April 23, 1976.
- Rubey, R. J., and R. D. Hartwick, "Quantitative Measurement of Program Quality," <u>Proceedings</u>, <u>ACM National Conference</u>, 1968, pp. 671-677.
- Brown, J. R., and M. Lipow, <u>The Quantitative Measurement of Software Safety and Reliability</u>, revised from TRW Report No. SDP-1776, August 1973, TRW Software Series (in press August 1976).
- Wulf, W. A., "Programming Methodology," Proceedings of a Symposium on the High Cost of Software, J. Goldberg (ed.), Stanford Research Institute, September 1973.
- 6. Abernathy, D. H., et al, "Survey of Design Goals for Operating Systems,"Georgia Institute of Technology Report GTIS-72-04.
- Weinberg, G. M., "The Psychology of Improved Programmer Performance," <u>Datamation</u>, November 1972, pp. 82-85.
- Kernighan, B. W., and P. J. Plauger, The Elements of Programming Style, McGraw-Hill, 1974.
- A Study of Fundamental Factors Underlying Software Maintenance Problems, CIRAD, Inc., December 1971.
- 10. Research Toward Ways of Improving Software Maintenance, CIRAD, Inc., January 1973.
- Merren, J., Software Portability, Stanford University Digital Systems Laboratory, Technical Note No. 48, September 1974.
- DeRoze, B. C., "DOD Defense System Software Management Program," <u>Abridged Proceedings from the Software Management Conference</u>, 1976 (obtainable through Los Angeles Section AIAA).
- Kossiakoff, A., and T. P. Sleight, "Software Requirements Analysis and Validation," <u>ibid</u>.

- Whitaker, W. A., "COD Common High Order Language (HOL) Program," ibid.
- Light, W., "Software Reliability/Quality Assurance Practices," <u>ibid.</u>
- Boehm, B. W., J. R. Brown, H. Kaspar, M. Lipow, G. J. MacLeod, M. J. Merritt, <u>Characteristics of Software Quality</u>, TRW Software Series TRW-SS-73-09, December 1973.
- 17. Thayer, T. A., et al. Software Reliability Study
  (Final Technical Report), TRW Report No. 762260.1.9-5, March 1976.
- Liskov, B. H., and S. N. Zilles, "Programming with Abstract Data Types," <u>ACM SIGPLAN Notices</u>, April 1974, pp. 50-59.
- Stepczyk, F. M., Requirements for Secure Operating Systems, TRW Software Series, TRW-53-74-05, June 1974.
- 20. Boehm, 8. W., R. K. McClean, and O. B. Urfrig,
  "Some Experiences with Automated Aids to the Design of Large-Scale Software." IEEE Trans. Software Engineering, March 1975, pp. 125-133.
- Teichroew, D. and H. Sayari, "Automation of System Building," <u>Datamation</u>, August 1971, pp. 25-30.
- 22. Alford, M. W., Jr., "A Requirements Engineering Methodology for Real-Time Processing Requirements."

  Proceedings, IEEE-ACM Second International Conference on Software Engineering, October 1976.
- 23. 3ell, T. E., and D. C. Bixler, "An Extendable Approach to Computer-Aided Software Requirements Engineering," ibid.
- Mills, H. D., Mathematical Foundations of Structured Programming, IBM-FSD Report 72-6012, 1972.
- Brown, J. R., Proceedings of the AIIE Conference on Software, Washington, D.C., July 19-21, 1976.
- 26. Fagan, M. E., Design and Code Inspections and Process Control in the Development of Programs, IBN- TR-21-572, December 1974.

# Appendix

# Definitions of Quality Characteristics

ACCESSIBILITY: Code possesses the characteristic accessibility to the extent that it facilitates salective use of its parts. (Examples: variable dimensioned arrays, or not using absolute constants.) Accessibility is necessary for efficiency, restability, and human arrangement.

ACCOUNTABILITY: Code possesses the characteristic iconumnability to the extent that its usage can be measured.

This means that critical segments of code can be instrumented with probes to measure timing, whether specified branches are exercised, etc. Gode used for probes is preferably invoked by conditional assembly techniques to eliminate the additional instruction words or added execution times when the measurements are not needed.

ACCURACY: Code possesses the characteristic accuracy to the extent that its outputs are sufficiently precise to satisfy their intended use. Necessary for

reliability.

AUGMENTABILITY: Code possesses the characteristic augmentability to the extent that it can easily accommodate expansion in component computational funtions or data storage requirements. This is a necessary characteristic for modifiability.

communicativeness: Code possesses the characteristic communicativeness to the extent that it facilitates the specification of inputs and provides outputs whose form and content are easy to assimilate and useful. Communicativeness is necessary for testability and human engineering.

COMPLETENESS: Code possesses the characteristic completeness to the extent that all its parts are present and each part is fully developed.

This implies that external references are available and required functions are coded and present as designed, etc.

CONCISENESS: Code possesses the characteristic concisences to the extent that excessive information is not present.

This implies that programs are not excessively fragmented into modules, overlays, functions and sub-routines, nor that the same sequence of code is repeated in numerous places, rather than defining a subroutine or macro; etc.

CONSISTENCY: Code possesses the characteristic internal consistency to the extent that it contains uniform notation, terminology and symbology within itself, and external consistency to the extent that the content is traceable to the requirements.

Internal consistency implies that coding standards are homogeneously adhered to; e.g., comments should not be unnecessarily extensive or wordy at one place, and insufficiently informative at another, that number of arguments in subroutine calls match with subroutine header, etc. External consistency implies that variable names and definitions, including physical units, are consistent with a Glossary; or, there is a one-one relationship between functional flow chart entities and coded routines or modules, atc.

DEVICE-INDEPENDENCE: Code possesses the characteristic device-independence to the extent it can be executed on computer hardware configurations other than its current one. Clearly this characteristic is a necessary condition for partability.

efficiency: Code possesses the characteristic efficiency to the extent that it fulfills its purpose without waste of resources.

This implies that choices of source code constructions are made in order to produce the minimum number of words of object code, or that where alternate algorithms are available, those taking the least time are chosen; or that information-backing density in core is high, etc. Of course, many of the ways of coding efficiently are not necessarily efficient in the sense of being cost-effective, since portability, maintainability, etc., may be degraded as a result.

HUMAN ENGINEERING: Code possesses the characteristic immon engineering to the extent that it fulfills its purpose without wasting the users' time and energy, or degrading their morale. This characteristic implies accessibility, robustness, and communicative-

LEGIBILITY: Code possesses the characteristic legibility to the extent that its function is easily discerned by reading the code. (Example: complex expressions have mnemonic variable names and parentnesses even if unnecessary.) Legibility is necessary for understandability.

MAINTAINABILITY: Code possesses the characteristic maintainability to the extent that it facilitates updating to satisfy new requirements or to correct deficiencies.

This implies that the code is understandable, testable and modifiable; e.g., comments are used to iocate subroutine calls and entry points, visual search for locations of branching statements and their targets is facilitated by special formats, or the program is designed to fit into available resources with plenty of margins to avoid major redesign, etc.

MODIFIABILITY: Code possesses the characteristic modifiability to the extent that it facilitates the incorporation of changes, once the nature of the desired change has been determined. Note the higher level of abstractness of this characteristic as compared with augmentability.

PORTABILITY: Code possesses the characteristic portability to the extent that it can be operated easily and well on computer configurations other than its current one.

This implies that special language features, not easily available at other facilities, are not used; or that standard library functions and suproutines are selected for universal applicability, etc.

RELIABILITY: Code possesses the characteristic reliability to the extent that it can be expected to perform its intended functions satisfactorily.

This implies that the program will compile, load, and execute, producing answers of the requisite accuracy; and that the program will continue to operate correctly, except for a tolerably small number of instances, while in operational use. It also implies that it is complete and externally consistent, etc.

ROBUSTNESS: Code possesses the characteristic robustness to the extent that it can continue to perform despite some violation of the assumptions in its specification.

This implies, for example, that the program will properly handle inputs out of range, or in different format or type than defined, without degracing its performance of functions not dependent on the non-standard inputs.

SELF-CONTAINEDNESS: Code possesses the characteristic self-containedness to the extent that it performs all its explicit and implicit functions within itself. Examples of implicit functions are initialization, input checking, diagnostics, etc.

SELF-DESCRIPTIVENESS: Code possesses the characteristic self-descriptiveness to the extent that it contains enough information for a reader to determine or verify its objectives, assumptions, constraints, inputs, outputs, components, and revision status. Commentary and traceability of previous changes by transforming previous versions of code into non-executable but present (or available by macro calls) code are some of the ways of providing this characteristic. Self-descriptiveness is necessary for both restability and understandability.

STRUCTUREDNESS: Code possesses the characteristic structuredness to the extent that it possesses a definite pattern of organization of its interdependent parts.

This implies that evolution of the orogram design has proceeded in an orderly and systematic manner, and that standard control structures have been followed in coding the program, etc.

TESTABILITY: Code possesses the characteristic testability to the extent that it facilitates the establishment of verification criteria and supports evaluation of its performance.

This implies that requirements are matched to specific modules, or diagnostic capabilities are provided, atc.

UNDERSTANDABILITY: Code possesses the characteristic understandability to the extent that its purpose is clear to the inspector.

This implies that variable names or symbols are used consistently, modules of code are self-descriptive, and the control structure is simple or in accordance with a prescribed standard, etc.

USABILITY (AS-IS UTILITY): Code possesses the characteristic usability to the extent that it is reliable, efficient and human-engineered.

This implies that the function performed by the program is useful elsewhere, is robust against numan errors (e.g., accepts either integer or real representations for type real variables), or does not require excessive core memory, etc.

#### MANAGEMENT OF SOFTWARE DEVELOPMENT \*

Edmund B. Daly .

GTE Automatic Electric Laboratories

# ABSTRACT

This paper describes five major aspects of software management: Development Statistics, Development Process, Development Objectives, Organization Structure and Software Maintenance. The control of both large and small software projects is included in the analysis.

#### INTRODUCTION

The concepts presented in this paper have been derived from the management techniques that were employed in three large software development projects. These projects accumulated nearly 2,000,000 hours of software development experience and required the generation of large real-time programs, extensive supporting programs and small minicomputer/microcomputer real-time programs.

The software projects completed development at three successive points in time. The latest project, capitalizing on experience gained during the two earlier projects, was developed exactly on schedule and 10% under budget. All available statistics indicate that the third project was a more efficiently managed project than the earlier two.

Although the data in this paper is historical, some of the proposed management concepts have not been used extensively, but are methods established to overcome earlier, less acceptable, management practices.

This paper is based on the following premise:

- Without a planned development methodology, software management will fail.
- Without an organization structure tailored to the selected development methodology, software management will fail.
- Without achievable objectives, formal controls and realistic estimating techniques software management will fail.

#### DEVELOPMENT STATISTICS

The unknown elements of software management can be reduced by employing historical development statistics as guide-

lines for estimating and controlling current development. Often these statistical guidelines are used by management to establish development objectives without spending sufficient time to investigate the foundation upon which the dara is generated.

# The Total Software Job

This section looks at the software development process from three new points: first, the total software job; second, the components of a software job; third, the dynamic flow of a software job.

GTE AEGo. has developed and cut into service three large stored program controlled telephone switching machines. The first project completed software integration in 1972 and incurred a development rate for online software of 3.0 hours per machine instruction. This program contained 160,000 machine instructions. The second and third projects completed development in early 1973 and late 1974. They encountered development rates of 2.3 hours per machine instruction and 1.9 hours per newly developed machine instruction, respectively. Each of these latter two projects were manned by completely different programmers, supervisors, and unit heads. Both of these latter projects required approximately 115,000 new machine instructions to be developed. In the above cases, development hours include all efforts required for specification, design, code, test and maintenance up to commercial availability of the program.

When looking at these statistics one may ask the following question: Is this development rate good or bad? The answer could be "bad" because software development rates can easily reach six minutes per source line; but then, the answer could also be "good" because some managers feel that trying to develop a program requiring more than 50 programmers is close to impossible. In order to compare the development rates of two different programs the following

\*Previously published in IEEE Transactions on Software Engineering, May 1977.

# factors must be considered:

- Design objectives
- Program size
- Program complexity
- Program language
- Program environment
- Data base
- Documentation standards
- Personnel and computer resources

# Design Objectives Affect Development Rates

A program can be designed to optimize one or more of the following standards:

Memory Utilization - Perform the required functions using the least amount of memory space to house both instructions and data base. This objective attempts to minimize system memory costs.

Executive Speed - Perform the required functions using minimum real time during execution. This objective accempts to maximize system through-put.

Schedule - Perform the desired functions in the least amount of time. In a competitive marker an "excellent" late product may be less profitable than an "acceptable" early product.

System Maintenance - Perform the desired functions so as to minimize on-site maintenance cost.

High Quality - Insure that the required functions are performed by well documented and thoroughly tested software. Design cannot rely on the customer to find the last 5% of software bugs.

Design Cost - Perform the required functions with minimum expenditure of resource dollars. This expenditure is in terms of manpower, material and travel. This objective should include both initial design as well as design maintenance.

Establishing quantitative values for these objectives is not a simple matter - at least not for the programs which are developed at a rate of 400 to 1000 source lines per man year. Unlike the programs which are developed at higher rates, these 400 to 1000 source lines per man year programs attempt to simultaneously optimize all the above objectives. To use an extreme example: One month the pressure is aimed at reducing memory size in order to meet competitive manufacturing costs. The following month generates pressures to either reduce development dollars due to smaller budgets or improve real time in a order to capture a larger market.

# Other Factors That Affect Development Rate

In addition to trying to optimize all objectives, the 400 to 1000 source lines per man year program usually encounters other manpower devouring requirements, the effort for which is usually buried into the development rate. These factors include:

- a. Large Data Base Requirements The development effort required to structure, define and document the data base.
- b. Large and Complex Program Requiring from 75,000 source lines
  to 500,000 source lines. These
  programs must pay the cost to
  integrate the output of many
  programming teams.
- Extensive Commercial Documentation
   Such as diagnostic dictionaries, input/output message manuals, user documents, charts, descriptions and extensively commented listings.
- d. Program Specified by An External Source - The design team does not have the option to decide what . feature requirements the program will satisfy - as such, these requirements are often loosely defined and constantly change during the early period of development.
- e. Generic Structure A single program must be able to work in many different environments the data structure must be designed to cope with varying requirements. Thus, the program remains constant and the data base information content and size changes from site to site.
- f. Complex Interfaces Program must be able to work with different quantities of hardware. The program also has extensive hardware /software interfaces as well as man/machine interfaces.
  - g. Software Responsible for System Reliability - Software controls are designed to minimize the effect on system operation due to hardware faults, hardware noise, data base errors or instruction errors.
- h. Maintainable Programs are designed to last twenty years with active maintenance occurring at least during the first four years. Good structured design is thus required.

# Resources Also Affect Development Rates

Real-time software development rates are very dependent on the resources available to management. Three important

#### . resources are:

- The programmer
- Support software (we will discuss only two support packages: compiler and simulator).
- Support computer.

Programmer - A Resource - In most applications a balanced team consisting of one leader, two experienced programmers and three inexperienced energetic programmers is ideal during the implementation and maintenance phases. A smaller team of experienced system level programmers is usually required during the earlier planning and specification phases.

Compiler - A Resource - A good compiler allows programmers to code in high-level language without extensive waste of memory. A 20% reduction in total development effort has been achieved at GTE AECo. when a realtime program was designed using high-level language rather than assembly language. This reduction in effort occurs in the following design phases: coding, documentation, testing, design maintenance. Because a source line written in high-level language contains more intelligence than does a source line written in assembly level language, a high-level language source line often incurs a larger "development cost per source line" than does the simpler assembly level source line. An assembly level source line usually produces one machine instruction. A high-level source line usually produces more than one machine instruction.

Simulator - A Resource - Large realtime programs must be eventually tested on
the machine in which they will reside. This
machine is called the "object machine". The
amount of object machine time required to
test a program depends heavily on the amount
of prior program testing which can be
performed off-line as well as the quality of
the object machine utilities. One method of
off-line testing is called simulation. The
process requires the design of a support
program called a simulator. This simulator
makes an off-line computer look like the
object machine; at least to the real-time
program which is being tested.

For large real-time programs, such as the ones described earlier, one hour of object machine time is required to test ten instructions of unsimulated, assembly level code. If simulation is used prior to object machine testing, this rate increases to twenty-five instructions per hour. The least amount of object machine time is required for simulated programs which were coded in highlevel language. These programs have been tested on the object machine at an average rate of forty instructions per hour.

The reduction in amount of machine time required for testing affects development expenditures in two ways:

- a. A programmer requires three hours of preparation and analysis time for each one hour spent testing on the object machine. Therefore, a one hour reduction in machine time carries a four hour reduction in development hours.
- b. Object machine time is very expensive. For the large real-time programs described in this paper machine costs range from \$40.00 to \$150.00 per hour.

Computer - A Resource - In addition to supporting software, the quality and quantity of computer resources affects efficiency of coding and testing. Currently, interactive editing and testing seems to be the most effective way to debug software. Batch, however, is still an effective development tool as long as turn-around time remains under 2.5 hours.

In conclusion, development rates should only be used as a guideline for planning new designs. These-rates cannot easily be compared - even for similar jobs - since management objectives or available resources may be different. In any case, the program incurring the more costly development rate may be the better design approach since it could require less memory and execute faster, thus requiring a less costly memory system and forcing a larger market. The object of software management should always be to optimize profits. This objective may require larger initial development expenditures in terms of hours per source line.

# Different Program Classifications

With this background, let us now look at the development rate for various types of programs in order to get some feeling of how development rates fluctuate. The range of development varies from 0.24 machine instructions per hour to 12 machine instructions per hour. I do not believe that the high development rate indicates better management but merely a different software job.

For simplicity let us categorize programs into three areas: Small Real-Time, Large Real-Time, Support.

# Small Real-Time Programs

Data has been compiled from five different minicomputer developments. These programs usually range from 5,000 to 20,000 machine instructions and range of effort varies from 1.6 machine instructions per hour to 5 machine instructions per hour. Intermediate rates are 2.3 machine instructions per hour and 1.9 machine instructions per hour.

The program which was developed at 5 machine instructions per hour did not require commercial documentation, nor was memory usage or real time a strong guideline.

Managerial objectives were to generate program in minimal time, requiring minimal man/machine interface. The system was designed to drive traffic through a larger machine. System up-time was not an important factor. The program was designed by one experienced programmer. The software was operational in three months.

The programs which were developed at a slower rate faced completely different design objectives and environments. They required large data base structures. Realtime execution as well as system up-time were important factors. These systems required commercial documentation and were designed for low cost software maintenance (i.e., modular construction).

# Large Real Time Programs

Chart 1 shows development statistics for three large real-time programs. All three programs were designed to meet the most stringent operational requirements. The most important single factor which determined the different development rates is the experience level of the designing programmers and supervisors.

## CHART 1

SYSTEM	COMMERCIALLY AVAILABLE TO CUSTOMER	SIZE OF PROGRAM (NEW INSTRUCTIONS)	DEVELOPMENT RATE INSTRUCTIONS PER HOUR
A	1972	160.000	.33
8	1973	117.000	.43 .
С	1974	111.000	.53

The above programs are constructed using many subprograms. On an average each subprogram houses 3,000 machine instructions or 20 software modules. The development rate for subprograms is different than for the total program in that some subprograms are very complex - such as diagnostic subprograms - and some are rather simple - pure data manipulation. The following list gives two examples of subprogram development rates.

- Data Manipulation such as updating memory information. 1.9 machine instructions per hour for subprograms.
- Diagnostic such as routining electronic hardware to determine if it is operational. 0.24 machine instructions per hour for subprograms.

In each case an additional effort of 5% is required to integrate each subprogram into a total program.

#### Support Programs

These programs are intermediate size (20,000 source lines). The programs referred

to in this section are called support software because they have been designed to "support" the design and manufacture of the large real-time programs described in the last section. Support programs consist of loaders, editors, assemblers, compilers, program generators, simulators and utility programs. They have been designed to execute on IBM-360/370 The size of support software is machines. large: 400,000 source lines of support soft-ware has been designed to "support" the designed manufacture of 370,000 machine instructhe design tions of real-time software. Support programs, however, are less complex to design than real time operational programs. Also, 60% of the code is implemented using high-level language. The development rate ranges from 0.7 source lines per hour for the more complex support programs to 12 source lines per hour for the simpler programs. The average development rate for the entire package of 400.000 source lines is 1.5 source lines per hour.

# Development Rate - Design Maintenance

Not only is the initial development effort associated with support software less (per source line) than that required for large real time programs but the design maintenance effort is also less. Statistics based on maintaining three separate real time software packages and two separate support packages indicate the following design maintenance effort:

- a. One equivalent programmer (two programmers each devoting half time to maintaining software) can maintain from 15,000 to 30,000 source lines of on-line real-time programs. This can be compared to the rate for support software of from 50,000 to 120,000 source lines per equivalent programmer.
- b. One programmer, devoting full time to maintaining software, can maintain 10,000 source lines of on-line realtime programs - or 30,000 source lines of support programs. This number differs from that presented in "a" in that we have found that design maintenance programmers are more efficiently employed if approximately 50% of their working hours are spent in new design.

The maintenance rates given above do not include such overhead items as configuration control, laboratory and field support, supervision.

# Development Rates - Components

In addition to specifying development hours per source line, another important development statistic divides this total development effort into its component parts: specification, design (includes code), test, documentation, design maintenance.

Chart 2 is a pie chart which shows this division for the three large real-time programs described earlier. Although,

Chart 2 represents an average of the three programs, each program taken individually indicates a similar distribution. However, Chart 2 does not include the effort required to "evaluate" the software package. This effort usually takes place at the first field site. For large real-time programs approximately 0.1 hours per instruction is required to write and execute evaluation test plans. The maintenance effort shown in Chart 2 is used to resolve software bugs found during evaluation.

CHART 2
DEVELOPMENT PIE CHART INCLUDING DESIGN
MAINTENANCE TO ONE YEAR AFTER CUTOVER

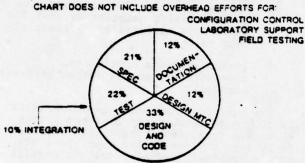
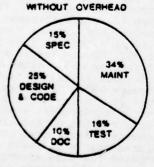


Chart 3 shows the same breakdown as Chart 2 but at a different point in time. Note that design maintenance effort has increased from 12% of total effort (one year after in service), to 34% of total effort after four years of service.

CHART 3

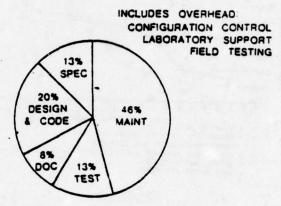
DEVELOPMENT PIE CHART INCLUDING DESIGN
MAINTENANCE TO FOUR YEARS AFTER IN SERVICE



Charts 2 and 3 include design maintenance effort only to the extent that it is performed by a programmer. This effort is only 55% of the total expenditure allocable to maintaining real-time software. The remaining 45% of total maintenance cost arises from prototype and field support manpower which is used to support the programmers testing activities, to evaluate his tested programs and to place these programs under configuration control. If we allocate these overhead charges to design maintenance, Chart 4 results. This chart indicates that design maintenance of software accounts for

46% of total software development dollar expenditure at a period of 4 years after commercial availability. Since the major maintenance effort takes place during the first four years after the commercial release of a program, the 46% should increase very slowly - possibly approaching 60% after 10 years. Design maintenance as used in Chart 4 includes laboratory efforts associated with fixing design problems and with small program enhancements. It does not include efforts required for large feature additions or local "on-site" support.

# CHART 4



# Dynamic Flow

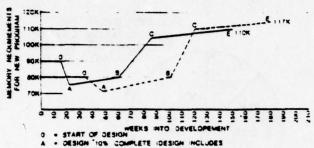
The software development statistics discussed to this point have been static statistics - taken at one point in time. Charts 5 and 6 show how a software project progresses through its various phases. These charts illustrate statistics for two different large real-time programs - developed by two different teams of programmers, supervisors and unit heads.

Chart 5 shows how estimated program size varies throughout the period of development. "Estimated program size" is a projection of the program size at commercial availability. A surprising "characteristic curve" is obvious from this chart. In both projects, program size estimates made early in development (start of specification) were high and then rapidly dropped - hitting a low ebb at the start of design. Estimated program size then starts to increase at a conservatively slow rate until design reaches 60% complete. At this point in time both projects increased estimated program size estimates at a very rapid rate - resulting in a size increase of about 50% from point A to point C. After design completion reached the 98%, program size estimates level off and were within 5% of the actual program size at commercial availability.

Chart 6 is an actual plot tracing the development of the 110,000 instruction real-time program, we see the progress of software development for specification, design, integration and evaluation. Note that in all cases there is a natural overlap of

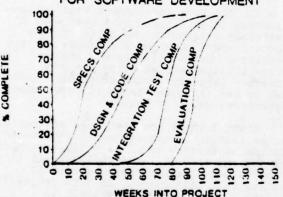
activities, in that at any one point in time many phases of development are taking place simultaneously. Chart 6 shows that the time required to complete the last 20% of a development phase is as long as the design time to complete the middle 70%. This characteristic is responsible for many schedule slippages since a good portion of the activity being performed during the latter 20% consists of correcting mistakes: either overlooked modules or unacceptable design.

## CHART 5 PROGRAM SIZE VARIATION - TWO PROJECTS -



- GODEL SPECIFICATION SOS COMPLETE
- DESIGN 50% COMPLETE SPECIFICATION 35% COMPLETE DESIGN 36% COMPLETE
- . SOOD WORDS PER YEAR

# CHART 6 CHARACTERISTIC 'S' CURVE FOR SOFTWARE DEVELOPMENT



# Development Statistics - A Future Trend

Although this section is concerned with software development rates - we can track the future evolution of these rates by comparing them to those being experienced in the related field of electronic hardware.

One of the large real-time programs referred to in this paper contains 160,000 source lines of software. This program controls the operation of a hardware system containing approximately 170,000 logic gates.

Historical statistics gathered in this system indicate that there is a close analogy between hardware and software development statistics. For example:

a. Twice as much effort is required to

- develop one source line of software than to develop one hardware logic gate.
- b. The same amount of effort is required to develop a large functional soft-ware segment as a functional logic board in hardware (containing 30 zates).
- c. Using the total number of system source lines (160,000) and total number of logic gates (170,000) as a base for comparison, then design maintenance corrections per source line exceed design maintenance corrections per logic gate by a factor of four to one. Design maintenance cost for software also exceeds design maintenance cost for electronic hardware by a factor of . four to one.
- d. System software complexity approximately equals system hardware complexity.

The question which must be answered is "why did total development cost per source line exceed total development cost per logic gate by such a large factor when the system hardware complexity equaled the system software complexity and the number of source -lines equaled the number of logic gates?" The answer lies in four areas:

- Hardware management techniques and development procedures were more advanced than those used in software.
- b. Hardware designers were more experienced.
- c. Hardware employed a more "structured ·design".
- d. The basic building blocks used in software design (i.e., different types of source statements) were more numerous and complex than the simple building blocks used in hardware: AND, OR, NOT. (Use of structured programming could eliminate this difference since the technique limits software building blocks to three types of single-entry, single-exit control structures: simple selection, simple repetition and linear sequence. Structured programming insures that a software source line contains the same amount of intelligence as a hardware logic gate.)
- e. In a software/hardware system the hardware gets a double dose of testing and evaluation: the first dose occurs during scheduled hardware testing, the second dose occurs as a natural byproduct of software testing.

In conclusion - as software management and design techniques approach the level that exists in hardware, and as software design becomes more structured, projects such as the one described above we can expect development rates per source line and maintenance costs per source line to decrease - hopefully approaching rates presently being experienced by an electronic logic gate.

# THE PROCESS OF SOFTWARE DEVELOPMENT

Statistical data gathered during the process of software development can be used to establish an efficient methodology for future development. For example, historical analysis indicates that over 50% of all development hours are spent correcting buss which result from faulty design. Information such as this tends to guide management toward development methodologies which place more emphasis in generating a higher quality initial design with the expectation that this additional early expenditure will be paid back during testing and design maintenance.

Since the primary output of software development is "code", many methodologies place heavy emphasis on generating code early in the development cycle. These software projects usually experience testing phases which last significantly longer than the combined specification, design and code phases. Many of the projects which rush the process of generating code are the same projects which experience the longest total development schedule. Chart 6 shows development completion curves for a large software project where generation of code was not given prime importance. Even in this project the testing and evaluation phase lasted approximately the same amount of time as the combined specification design and code phases.

Chart 7 shows two "extreme" approaches to software development. More experienced development organizations are tending toward Method 1 since it produces a higher quality software product at a lower development cost-especially if one considers long term maintenance.

Method 2 seems to be a more natural approach in that it requires less management control, requires less technical (software) experience, allows programmers to both innovate and avoid what they have most documentation.

Method 2 will also require less memory space and less execution time than Method 1. The reasons for this fact are threefold:

- Structured design requires extensive subroutine linkage. This process utilizes extra memory and execution time in order to maintain integrity of "computer register" information. (Some minicomputers are being designed to minimize this overhead by saving the contents of program register via hardware at each subroutine call.)
- High-level language requires more memory space and execution time than assembly level implementation of the same job. Even the better compilers require 10-15% more memory space and execution time. This percentage

increases to 100% for many commercial high-level languages.

- The tight complex code employed in Method 2 will require less program storage space and execution time than the structured code used in Method 1.

A good guideline is to employ a methodology somewhere between the two - but much closer to Method 1. The three large GTE projects referenced in this paper followed this guideline.

# CHART 7 SOFTWARE DESIGN METHODS

METHOD 1 METHOD 2

HIGH-LEVEL LANGUAGE.
STRUCTURED CODE.
COMPOSITE DESIGN
(HERARCHY OF SMALL
SEGMENTS).

PARALLEL: TOP DOWN. BOTTOM UP DESIGN ALL OPTIONALLY USED

SIMPLE DATA STRUCTURES AND WORK AREAS (NOT TIGHTLY PACRED) TEAM APPROACH TO DESIGN (EGOLESS PROGRAMMING)

:8M'S STRUCTURED WALK THROUGH FOR REVIEWING DETAIL DESIGN AND CODE. ASSEMBLY LANGUAGE.
TIGHT COMPLEX CODE
LARGE BLOBS OF CODE.

BOTTOM UP DESIGN

TIGHT EFFICIENT DATA STRUCTURES AND WORK AREAS (LVEHY- HIT USED NO DATA DUPLICATED)

"ONE PROGRAM ON MAN CONSETS

NO DETAILED TECHNICAL REVIEW OF DESIGN OR CODE.

# CHART 7 SOFTWARE DESIGN VETHODS METHOD 1 METHOD 2

THREE SEPARATE TEAMS
ONE TEAM JESIGNS ONE
TESTS. ONE EVALUATES
COMMETTE SET OF HERARCHY
CHARTS. SEQUENCE CHARTS.
DATA MAPS AND NARRATIVES
WELL COMMENTED LISTINGS

DETAILED TEST PLANS FOR ALL TEST PHASES. PROGRAM MAINTAINED BY 30% SENIOR PROGRAMMERS ONLY COMMERCIAL DOCUMENTATION SENERATED DURING DEVELOPMENT

STRICT MANAGEMENT OBJECTIVES ESTABLISHED TO GLIDE DEVELOPMENT

ORIGINAL CODER TESTS. INTEGRATES AND HELPS EVALUATE HIS PROGRAM

DETAILED FLOW CHARTS AND

NO CONSISTANCY IN LISTING COMMENTS

NO FORMAL TEST PLANS

PROGRAM MAINTAINED BY INEXPERIENCED PROGRAMMERS OR TECHNICIANS EXTENSIVE NONCOMMERCIAL TECHNICAL MEMORANDUM GENERATED AND PLACED ILBRARY

NO MANAGEMENT OBJECTIVES

### A WORKABLE DEVELOPMENT METHODOLOGY

The following section describes a management approach which is being used by some newer projects in GTE AEL to generate commercial software. The process which is described is not a revolution in management techniques, but standard management practices melded with the idiosyncrasies of software development. These techniques have evolved during the development and maintenance of GTE's EAX and TSPS stored program switching machines.

There is no universal process which should be used to develop all types of software. Large commercial real-time programs which involve more than 30 programmers require a development methodology containing more documentation, more cross-checks and in general - better management techniques than do smaller non-commercial programs that may only be executed in a few machines and usually by highly qualified engineers. What is important is that management establish a development process for each type of software design - early in the planning stage. The methodology presented here is applicable to

large real-time commercial software packages. - and will result from a combination The process can be modified for "simpler" of poor supervision and substandary developments by eliminating some of the less critical documents and technical cross-checks.

# Development Cycle

There is a specific cycle associated with the generation of commercial software. This cycle can be divided into five phases: plan, specify, design, code, test. Although these phases are somewhat universal for all software developments, the specific management methods used to insure successful completion of each phase differs due to variations in management styles, and more importantly, variations in management maturity.

Before looking at each of the above development phases in detail, there is a list of empirical management guidelines which seem to be applicable in all large software developments.

- a. Management review of development progress will not insure successful completion. Management must expect not to be told the completely story - especially in trouble areas. Thus, detailed technical cross-checks are necessary.
- b. Pert-Cost or any scheduling and control method should be used only as an aid to management. Far too often, management uses formal techniques as a substitute for "real" management. At best, techniques such as PERT tell only real" part of the story - usually the better part. However, if used properly, Perc-Cosc, or its equivalent, is a necessary management tool for controlling large software developments.
- c. Management cannot dictate unnatural schedules and expect a working. maintainable program to result. Poor programs are often a direct result of either inexperienced personnel or forced schedules.
- d. Because the early phases of software development have no visable output except documentation, major milestones must be associated with a completed documentation package.

A defined milescone with no visable output is a useless milestone for management control. In order to insure proper reviews, documentation required for each major milestone must be defined in detail before development starts.

e. Management should expect that 5% of all software development will not work at all and must be redesigned.
It will be poorly documented and employ unacceptable design tech-niques. This fact will be true regardless of published standards

of poor supervision and substandard programmers.

These unacceptable programs will not be detected by either management reviews or scheduling techniques such as PERT.

Management must therefore establish techniques to find these "bad" programs before testing begins in order to insure minimal effect on project schedules and development COST.

f. Software development costs can be minimized by limiting the generation of technical documentation to that which will be maintained for the life of the software program (20 vears).

Many development projects generate technical documents which are not maintained. These documents are not only expensive to generate but soon become useless and outdated; or even worse become dangerous, since they contain incorrect technical detail.

- g. Once a program is working in the field, changes should never be made unless absolutely necessary. Arbitrary enhancements or modifications induce software bugs and thus should be avoided.
- h. Software costs can be minimized by finding design bugs early in development. Using IBM's concept of "Structured Walk Through", most design bugs can be found prior to testing. The cost of software bugs increases significantly after a program responsibility passes from the initial designer to a design maintenance group.

A technical review of detailed code is required prior to machine testing. This review can be performed by the programmers immediate supervisor, a chief programmer (if teams are employed) or by a small group of peer programmers.

i. Management cannot assume that programmers know how to design software properly. Many experienced programmers (and most inexperienced programmers) tend to generate tricky, complex, tight, hard to understand, and poorly documented

Detailed design and coding standards must be established and followed in order for the resultant program to be maintainable at a reasonable cost.

j. Design Maintenance programmers require a higher level of experience than that required for original

design. Historically management tends to place less experienced programmers in design maintenance. This is a costly and dangerous mistake.

# A Development Process

Using these and other empirically derived guidelines a software development process has been derived. This process, which covers all phases of development from initial planning to design maintenance is summarized in Chart 8.

The process recognizes the need for two types of design reviews: technical and managerial. Managerial reviews are held at major development milestones. The object of these reviews are three-fold: review commercial documentation which is scheduled for completion at the specific milestone; review development expenditures and schedules - both historical and forecasted; approve continued development. Technical reviews are much more detailed than management reviews and do not consider schedules or budgets. The primary responsibility of a technical review is to analyze the same commercial documentation presented to management but in a very detailed manner. Technical reviews are always held prior to management reviews and must be successfully completed before the management review convenes. To illustrate the difference between these two reviews we will use the completion of "coding" as an example of meeting a major milestone.

Before looking at the contents of a code review it is important to understand how a program is constructed. Programs are divided into successively smaller sections called subprograms, modules, segments. This division into simpler parts allows a complex program to be defined, developed and maintained. Whereas a large program may require 50 designers, a large subprogram may require 5 designers. A module is usually assigned to one designer. As a further simplification, the designer divides his module into individual functions called segments. Segments are then coded.

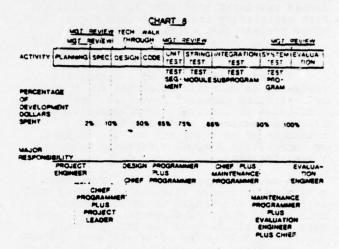
In summary, programs are built using subprograms. A subprogram performs a major system activity. An average subprogram consists of about 3,000 source lines. Subprograms are built using modules. A module performs a set of very closely related functions and consists of about 300 source lines. Modules are built using segments. A segment performs one function and consists of 50 source lines. Segments are built from source lines (or instructions). Source lines are the most basic building block in software design.

# Coding Review

Experience indicates that this review should take place at about the point where 65% of all software development dollars have been expended. In this case, total software

development dollars include all expenditures for planning, specification, design, code and test.

Since the completion of coding is a major milestone, it requires both, a technical and managerial review. As shown in Chart 8, the technical review comes first. This review is organized to detect most "bad" code. The resultant code should be simple, straight forward and easy to understand. Where applicable, structured coding techniques should also be employed.



Unit tests should be made available at the code review and a subset of these tests "mentally" executed by the reviewing body. This process, when combined with code reading (a process where code logic and code format is scrutinized by a programmer other than original designer), can detect up to 90% of coding and syntax errors. Code reading and mental testing should always be employed prior to object-machine testing in order to minimize usage of expensive machine time. Development cost required to detect an error by reading code is approximately equal to 25% that required to detect an error via machine debugging.

Probably the most important concept associated with this technical review is the actual commercial code is reviewed. The flow and structure of the code is analyzed for design flaws and mis-interpretations. The review body is kept small - always including the chief programmer. In some reviews additional people may attend: coders who are responsible for software directly interfacing with the module being reviewed, first level supervisor and the programmer who will be responsible for maintaining the software module after development is complete.

Once the technical review is complete, management should be given a presentation summarizing the results of the preceding reviews: the technical review held after completion of design and the technical review held after completion of code.

The management review is held after completion of coding rather than after completion of design so that more accurate instruction estimates will be available (see Chart 5). Since these reviews are held on a subprogram basis, the first review will be held the early part of the "design and code complete" curve shown in Chart 6. It is good practice to require test plans and all commercial documentation to be brought to the management review - for psychological purposes only. Management should be concerned resolving non-technical problems and with correlating results of this review with other project reviews (a large software project will have coding reviews spread out over a year or more).

The major objective of a management review is to maintain schedules and budget by: shifting manpower from less important activities to critical tasks, cancelling or delaying features, allowing standard practices to be short-cut and if all fails to immediately publish a schedule slip or budget increase.

A second objective of the management review is to insure that the project is still profitable. It should be noted, however, that 65% of total development dollars have been spent by the time the coding review occurs. At this stage of development, customer commitments have probably been solidified to such an extent that even unprofitable projects continue to completion.

The management review process which occurs after code is complete is probably the most comprehensive as far as technical content. Earlier management reviews concentrate more on "should the project continue". For this reason management reviews are concentrated around the specification stage.

#### THE DEVELOPMENT PROCESS

This section will describe a typical flow through the development cycle. The description is somewhat detailed but this emphasis is justified since the development process is the heart of software management. If a "good" development process is combined with a "good" scheduling and cost control tools, management will experience a high percentage of successful software projects.

## Phase 1 - Planning

The first phase of development is called the planning stage. During this stage the customer generates his requirements. Normally the customer will work directly with project engineers. If a project requires only software effort, this engineer should be an experienced systems programmer. If a project requires hardware and software, the project engineer must be knowledgeable in both areas.

A primary objective of planning is to quickly generate a "high level packaze" for management review. This phase should not spend more than 2% of total development

dollars. It is at this first review that most marginal or non-profitable projects will be "turned off" by management.

The types of information brought to management by the project engineers are:

- Product description description of system features as seen by the customer.
- High level description of how major features will be implemented.
- Software hierarchy chart where each element in the chart represents about 3,000 instructions (a subprogram).
- Estimates of schedule, development cost, memory size, real time, market potential, material and system cost.

Armed with the above information, management can determine if the reviewed project deserves further funds. The estimates vary in accuracy depending on the completeness of the original product description and the estimaters' experience. + 35% seems to be an average accuracy figure. Chart 5 indicates that management can improve upon this accuracy by increasing the program size estimation which they receive from their technical staff by 30%.

If management approves further development, a project leader is appointed. Chief. programmers are also selected at this time. These people will execute the next stage. The project leader coordinates all project activities - both hardware and software. He is responsible for integrating schedules of all involved development groups. The project leader also controls all system level documents - such as the specification of how the system (or program) is to operate and what features are included. The chief programmer is a technical leader who will eventually be in charge of a team of programmers. Large projects will have one project leader and many chief programmers (a chief programmer team should be limited to five programmers). In small projects. requiring only one chief programmer, the project leader and chief programmer can be the same person.

#### Phase 2 - Specification

The next management review will occur after 10% of the total development dollars estimated during Phase I have been expended. The output of this phase is generated by the project leader and chief programmer(s). All technical documentation is generated in a format acceptable for commercial release.

Since the project is still in early phases of development, the probability of managerial rejection is high. During this phase, firm marketing estimates are established. Cost and schedule accuracies are improved from 35% to 15%. The primary object of this phase is to bring accurate information to management for project

approval of disapproval. The probability that a project will be approved at this stage and turned off at a later stage should be small, for after the 10% phase development costs increase very rapidly and a total manpower commitment is made.

The output generated during the specification stage consists of the following technical documents:

- Subprogram hierarchy chart showing control flow between modules.
- Time sequence description explaining the sequence in which modules execute in order to perform major system features.
- c. Subprogram prologues which describe the functions performed by each subprogram, the inputs to each subprogram and the outputs from each subprogram.
- d. Module prologues which describe the functions performed by each module, module inputs and module outputs.
- A general layout and description of all data used by the program.
- A description of all man/machine interfaces.

The administrative documents generated during the specification stage commit to specific values for: memory size, real time, development cost, development schedule, and resource requirements. These technical and administrative documents form a "contract" by the developing organization.

# Phase 3 - Design

After Phase 2, development activities accelerate very rapidly. If the technical base established in Phase 2 is generated properly, a significant number of programmers can be placed on the software problem early in Phase 3.

The Phase 3 review occurs after 50% of total development costs specified in Phase 2 have been expended. Phase 3 is a turning point in that most managers no longer have the ability (or time) to perform a detailed review of work performed and thus must rely on technical "walk-throughs" (IBM terminology). The output of Phase 3 contains no code. This is an important concept. The walk-through is a formal review technique where test inputs are generated and "eyeballed" through the design to verify its accuracy. Experience shows that coding is started far too early in most software projects and because of this a large percentage of code is redone before the software passes evaluation.

The technical "walk-through" performed at the conclusion of Phase 3 is attended by the project leader, (he is responsible for

program specification), chief programmer, interfacing programmers and the supervisor. The output of Phase 3 censists of:

- An update of all technical documentation - described in Phase 2.
- Module hierarchy charts which show control flow between segments.
- c. Prologues which describe the function performed by each segment within a module. Definition of all data inputs to each segment and data outputs from each segment.
- d. Test plans required to test the operation of each module.
- e. Detailed data maps and description of each table and item.

All these documents with exception of module test plans are generated for commercial release.

This review is organized to detect all "bad" program modules. "Bad" in the sense of poor structure. We require programs to be built from code blocks - called segments. Each code block should perform only one function and be fully documented; program modules are built from these code blocks using rules of structured design. "Bad" in the sense of incorrect or pooly defined interfaces; modules are also reviewed to insure correct inter-module interfaces and to assure that functions performed are in agreement with program specification.

# Phase 4 - Coding

This is the coding phase that was previously explained. Since the entire program is completely specified after Phase 3. segment coding can take place in parallel manner. Top down coding need not be employed. Top-down coding will be used only in those developments where manpower availability indicates that top-down will not delay project completion.

From an administrative standpoint, all estimates are refined. An accuracy figure of better than 10% should be available for memory size, real time requirements and development costs. An accuracy of better than 5% should be available for schedules.

# Phase 5 - Testing

This phase consists of four stages of software testing.

First, each segment is tested to insure its singular function works as specified in Phase 3. Second, the segments are strung together to insure that groupings of segments (called modules) work as specified in Phase 3. Both of these testing activities are performed by the design programmer. Either top-down and bottom-up testing may be employed. Modules which are tested earlier

are high-level control segments, hardware interface segments or segments prone to interface errors.

After the chief programmer agrees that unit testing and string testing is complete, the chief programmer together with the maintenance programmer (the person who will be responsible for maintenance) verify the operation of each subprogram. The maintenance programmer being an experienced programmer is responsible, along with the chief programmer, for zenerating (or at least agreeing with) the integration test plan which checks the subprogram operation. The generation of integration tests should start as soon as possible after Phase 2. Early development of these test plans helps uncover "holes" in the software specification.

Since the maintenance programmer and the chief programmer should report to different first line supervisor, an important crosscheck is established. At completion of integration testing, the maintenance programmer must agree that the subprogram works according to the specification generated in Phases 2 and 3. The maintenance programmer also formally agrees that the subprogram meets commercial standards: well documented, structured design, easy to understand code. Because the maintenance programmer will be responsible for the subprogram after the original designer relinquishes responsibility, he will not accept the subprogram without serious analysis. Unacceptable subprograms are immediately brought to the attention of second level managers and the project leader.

The last stage of design testing checks the inter-operation of all subprograms. A team approach is used for this "total program" or system test. The team consists of selected chief programmers, senior maintenance programmers, and evaluation engineers. All members are involved in generating and executing the test plan. The evaluation engineer is a system expert. In those areas where hardware and software are involved, the evaluation engineer should be knowledgeable. from a system standpoint, of all areas. During system testing, all functions of the program are tested under stress conditions. Errors are placed into operational software, data structures and system hardware. External drivers are employed to simulate heavy "user" activity. The objective of system testing is to make the system fail.

At the conclusion of system testing, the evaluation engineer must formally agree that the program successfully passed system testing. This acceptance is accompanied by a list of outstanding problems. Acceptance is contingent upon these problems being resolved. The program list is passed on to second level management and the project leader to insure prompt resolution. A formal follow-up procedure must be established to give these problems appropriate priority. During system testing, and for the life of the program - all software bugs are formally documented and formally scheduled for resolution. Summary data and dicating "bugs per module" is fed back

to design supervisors. This data allows 'supervisors to evaluate and improve design procedures as well as programming staff.

After system testing is complete, the responsibility for modifying the source program is relinquished by the design programmer (and chief programmer) and passes on to the maintenance programmer. After system testing, the program should work in its intended environment. The software maintenance phase now begins and the total system is evaluated by an independent group of system experts. The primary function of "evaluation" is to insure that the system meets all customer requirements from a feature standpoint, a man/machine interface standpoint, a documentation standpoint, and real-time standpoint. During evaluation, a complete set of acceptance tests are executed. Design problems are resolved by the maintenance programmers. The effort expended in evaluation testing is not considered part of the software development cost.

### DESIGN MAINTENANCE AND CONFIGURATION CONTROL

The successful development of a large software controlled system is at best only a partially completed task. Although many feel that the management of large software development is mysterious, or at least little understood; the long term control and maintenance of large programs is even more mysterious.

# Configuration Control

The management process which is used to maintain a program after completion of design is called "software configuration management"

This process allows management to control the maintenance and manufacture a software package (or program). The process also allows management to control modification to this package and to insure software changes are made coincident with associated hardware changes. Programs should be placed under configuration management after subprogram integration but before completion of system testing. After configuration control takes effect, all code and documentation changes must be accompanied by a supervisory approved form indicating reason for change and a test plan, if applicable.

The controlled software package is sold to the customer. It is called a "program version". If different sections of the market require slightly different programs - each section is supplied a different program version. One of the large real-time programs described earlier exists as four different versions. Two of the versions serve the same market but contain different feature packages. The other two versions serve different markets (outside USA). Each of the four versions contain approximately 1,000 modules. 80% of the modules are common among the four versions.

Given this situation, configuration management allows us to:

- Maintain the common modules only once. Thus 1,600 modules need to be maintained rather than 4,000 modules. A much larger design maintenance effort would be required if common modules were not recognized.
- b. Design new versions using existing common modules.
- Insure that corrections made to modules in one version are reflected in all versions.
- Allow systems to retrofit from one version to another version without affecting customer service.
- Insure that all software changes have proper management approval and are thoroughly tested before being sent to the field.

Under configuration management a version is updated either annually or semi-annually by issuing reassembled program loads. this is done, each size employing the specific version must place the updated load in its machine. This update is called a release. If important changes must be made to the commercial program at a faster rate than releases are generated, then patches may be sent to the field in the form of point releases. Experience indicates that point releases occur at six week intervals. Charts 9 and 10 show releases of the four versions of the above mentioned program.

A release contains both design corrections and new features. Versions remain active (i.e. are updated with releases) for approximately four years. During this period releases to these versions occur at approximately the following intervals:

- Three month interval for first six months.
- Six-month interval for next eighteen months.
- One year interval for next two years.

Controlling, approving and monitoring software fixes to a commercially released program is an important aspect of design maintenance. This function is most efficiently performed by a review group consisting of members who are close to both technical detail and management philosophies. Design maintenance supervisors are excellent candidates for this group. This group both schedules manpower to resolve software bugs and insures that reported problems are not enhancements to the original software specification.

The above controls are strictly enforced for non-critical problems. These controls must be relaxed when serious service affecting software problems are suspected to be in a released program. To resolve these problems in large software/hardware systems

a specially trained team (called an ATTACK TEAM) must be established. This group of specialists is given specific, system level, training aimed at quickly resolving complex problems which exist in an on-line site.

# CHART 9

RELEASES MUST NOT OCCUR AT SAME TIME SO THAT PROBLEMS ARE FOUND ONLY ONCE.

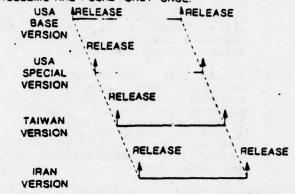
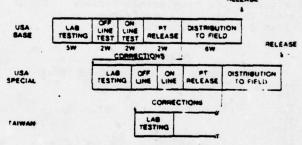


CHART 10

PELEASES SHOULD BE TESTED THREE TIMES WHENEVER POSSIBLE



- TRY TO MCK ON-LINE SITE WITH HEAVY REAL TIME REQUIREMENTS.
   TRY TO MCK SITE THAT REQUIRES ALL NEW FEATURES.
   TRY TO MCK FRIENDLY SITE

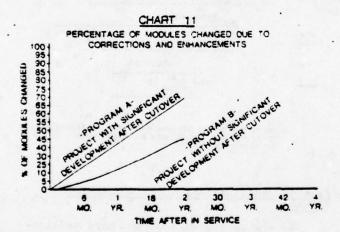
# Program Modifications After Commercial Release

As was mentioned earlier, more development effort is spent in the 10 years following a development than during the initial development. This characteristic does not seem to be particular to a specific application or to a specific design approach. Programs require extensive maintenance after the first commercial release for four reasons:

> Changes in Man/Machine Requirements - After a program reaches the field, customers find more effective ways to interface with the program either for maintenance or operacional reasons. Implementing changes is considered part of Implementing these design maintenance.

- b. Small Feature Additions and Enhancements As a program matures in the field, potential customers require added small features or enhancements to meet their specific environment. Since this activity is necessary for continued sales it is considered part of design maintenance.
- c. Latent Software Bugs No matter how carefully a program is tested prior to the first commercial release, a significant number of software bugs will be found in large real-time programs as the program faces different user environments.
- d. Induced Bugs New software which enters the field always contains new bugs. Analyzing the source of software bugs found in a commercial program indicates that 50% have always been there, 60% are due to new features, 15% are generated by fixing other software problems, 15% are miscellaneous.

Chart 11 shows the percentage of program modules which were modified for one of the above four reasons during the first two years after commercial introduction. Program B has seen 45% of its modules change during the first two years. This particular system has had few feature enhancements added to its base during its maintenance period. As can be seen in Chart 11, module changes to program B are tending to level.



Program A has seen over 70% of its modules change during this same two year period. This program has encountered significant feature enhancements and has also enjoyed a large market - which has forced the addition of many small enhancements to meet customer requirements and has forced the program to face many different external environments.

Since many modules are being modified during the early period of commercial availability, configuration management must enforce both tight control of software changes as well as extensive testing of these

changes before they are released to the field.

The configuration management controls call for four levels of testing of all program modifications before they are released to the field in the form of a version. As shown in Chart 10, these tests include:

- Original designer test (not in chart).
- Evaluation test on laboratory model for five weeks.
- Field test on commercial off-line site for two weeks.
- Field test on commercial on-line site for two weeks.

Chart 12 shows statistics indicating the number of program parches that were made to each of two different systems. These changes were sent to the field in the form of point releases. As can be seen in this chart, a formal test plan is required per software change. The chart also shows that all four levels of testing are required in order to achieve a program acceptable for field distribution. In Chart 12, System 3 did not test patches at an off-line site. Because of this, eight coding problems were found at the on-line test site. System A did have an off-line test site where most of the software bugs were found and corrected. Since these statistics have been compiled a fifth level of cross-check has been added to the above process. cross-check occurs after original designer testing. A separate group now visually inspects and approves all source updates and all test plans prior to initial evaluation testing. This group has been able to reduce the number of errors reaching evaluation testing by 40%.

CHART 12
FIELD STATISTICS

SYSTEM	NUMBER OF PATCHES	BAD CODE IN LAB	BAD CODE IN OFF-LINE SITE	BAD CODE IN CN-LINE SITE
A	550	23	7	
8	475	20	•	3
SYSTEM	NUMBER OF PATCHES	BAD TEST PLAN IN LABS	BAO TEST PLAN IN OFF- LINE SITE	BAD TEST
A	550	130	22	7
8	475	35	300 - 0000	11

## Software Sugs Found During Design Maintenance Are Costly

Early phases of software debugging are aimed at finding simple bugs. Later stages of testing usually concentrate on the more complex problems. Code reading is an example of testing which is executed very early in

the debugging phase - even prior to initial machine testing. The more complex software problems, however, usually show up during system testing or after the program has been placed into service. The development cost required to detect and resolve a software bug after it has been placed into service is thirty times larger than the cost required to detect and resolve a bug during the early "code reading" phase.

This large cost difference cannot be explained solely by the fact that the problems involved are of different complexity, for if we compare the development cost required to solve problems of equal complexity - the difference is still fifteen to one. This concept points to the general management guideline that software bugs should be found at the earliest possible stage of testing in order to minimize overall development costs and development schedules: it is more efficient to find and correct a simple coding mistake during early code reading than during machine test (by a factor of four to one) or during design maintenance (by a factor of fifteen to one).

Software bugs cost more to correct after a program has been released to the customer than during early phases of testing for the following reasons:

- After commercial release problems are usually more complex.
- b. After commercial release problems are reported as system malfunctions; an effort must be spent to translate problem into a software bug.
- c. After commercial release many problems are resolved by design maintenance programmers rather than the original designer. Design maintenance programmers must spend effort reviewing detailed code.
- d. After commercial release problems require more definition and more formal documentation. Formal test plans and multi-level testing must be performed to insure that accurate corrections reach the field.
- e. After commercial release problem resolution must share the heavy overhead cost for configuration management.

# ECONOMIC ANALYSIS OF SOFTWARE DESIGN

It is imperative that all software development be aimed at one goal - to maximize profits for the supporting company. Often this objective is lost among a complex entanglement of technical goals. To achieve the objective of profit maximization, four factors must be considered: Development and maintenance expenses, sales dollars, system cost, availability. Since costs are constantly changing due to inflation and technological advances in hardware

(especially solid state memories) projected costs rather than current costs should be used when calculating all expenditures and income.

Most software developments are both initiated and completed without managerially established objectives. Some of the better managed software projects are developed with such management objectives as: maximize real time or minimize storage. Although these technically oriented objectives are better than none at all they usually fall short of meeting the universal objectives of all managers - optimization of profits.

A large void currently separates software management and the concept of profit optimization. In most cases the reason for this gap exists because too few software leaders are trained managers and, inversely, too few good managers understand the specifics of software development.

Unfortunately, there is not one universal way to develop software. Contrary to the belief of many managers, all software need not be modular, structured or generic. We can all think of many examples. One obvious example is a program which must be designed by inexperienced programmers and used only one time - for a special on-line retrofit process. A good manager must not only be aware of all technologies associated with software development, but he must also take every opportunity to "test the water" by experimenting with development concepts. Thus, with experience, a good manager will be able to match various concepts of software development with specific project requirements and available resources.

Certainly, in an idealized world, software should be developed using high-level language. Software architecture should be modular and code should be structured. Documentation should be simple yet effective. Chief programmer concepts and egoless-programming should be practiced. Extensive integration and evaluation test plans are a must.

In the real world, however, these idealized concepts must be balanced against the fact that: compilers may not exist, programs may only be executed once then disregarded, talent mix may not justify a chief programmer. Time and budget may not allow extensive evaluation.

Thus, software management is like all other forms of management: it can be effectively practiced only by those individuals who have become both, mature technologists as well as mature managers.

In order to limit the scope of this paper I will consider the "profit optimization" concept for a specific type of software: a program which executes in realtime, controlling the operations of a large commercial hardware machine.

For this program, profit optimization can be met by first assuming a constant sales price and then by establishing "dollar value" trade-offs between the following controllable factors:

- Memory size (affects system cost).
- Real time (affects sales dollars).
- Development schedule (affects sales dollars).
- Development hours (affects expense).
- Design maintenance (affects expense).
- Machine maintenance hours (affects sales dollars).

A dollar value can be assigned to each factor. Management's task is to minimize cash out-flow (expense) and to maximize cash in-flow (sales dollars less cost of sales).

As a simple illustrative example of the above optimization approach let us answer the question: Should high-level language be used to develop the control program for a switching system?

Typically this question is answered by a manager, either affirmatively - because he feels that high-level language is good, or negatively - because he feels high-level language is bad.

To use the profit optimization process a manager will gather the following information:

- a. Memory Size High-level language will increase storage estimates from 100,000 words to 120,000 words. Cost of memory is 25c for each word. Therefore, cost of using high-level language is \$5,000 per system sold. The market life is one year and 100 systems will be sold. The total added cash out-flow for additional memory is \$500,000.
- b. Real Time High-level language will reduce available time by 10%. This reduction can be translated into a reduction in sales. As a first approximation a linear reduction in sales volume can be used (10% reduction). This reduction in sales represents a reduction in net cash in-flow (sales dollars less cost of sales) of \$100,000.
- c. Development Schedule Use of highlevel language will reduce development interval by 6 months. This means that the product will be available 6 months earlier. This data should now be translated into additional sales and consequently additional cash in-flow. As a first approximation assume that active sales life will increase by 6 months. With constant annual sales

cash in-flow will increase by 50% or \$500,000.

- d. Development Hours High-level language will decrease original design hours by 15%. Assuming \$20.00 per design hour and 2 hours per assembly level instruction, development cost, or cash out-flow, is reduced by \$600,000.
- e. Design Maintenance Hours Design maintenance effort is reduced by 5%. This calculates to a reduction in cash out-flow of \$100,000 (time value of money must be considered).
- f. Site Maintenance Hours Maintenance activity on each site is not affected. This area becomes important when considering whether improvements should be made to man/machine interfaces or whether improvements should be made to software diagnostics.

A profit optimization decision is made by adding "a" through "f".

Net advantage of high-level language

- = a + b + c + d + e + f
- = -500K -100K + 500K + 600K + 100K
- **=** \$600.000

Thus, high-level language, if available, should be employed in this application. If, however, a compiler must be designed - its development cost must be less than \$600,000.

### ORGANIZATION STRUCTURE

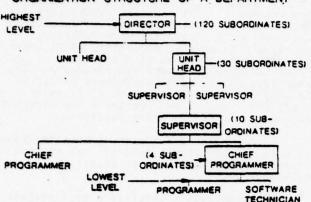
The structure of an organization can be analyzed from many viewpoints: What may look like a functional organization at a supervisory level in the organization structure (see Chart 13) may well look like a project organization from a unit-head level in the organization. Thus, in order to discuss organization structure one must specify which level in the organization is being used as a reference point.

I would like to start this discussion using a "department" or "director" level as reference. In this discussion a department is defined as a group whose size varies between 75 and 250 people. Ideally a department should contain no more than 120 professional people. It is controlled by three basic levels of management which we shall call supervisor, unit head, and director.

A department is chosen as the primary reference point since it is the highest level in a corporate structure which exerts a strong influence on the day to day work activities of programmers. As such, the three levels of department management have primary responsibility for budgets and schedules of activities being performed by the technical personnel within the department.

# CHART 13

# ORGANIZATION STRUCTURE OF A DEPARTMENT



Software, as with all other forms of development, should start with a controllable and stable organization structure. The selected organization will depend on many factors. Some of the more important factors are:

- a. Size of Each Software Development -Number of programmers whose output must be combined to make up one working program.
- b. Number of Projects Few large projects (30 or more programmers) or many small projects (10 or less programmers).
- c. Scope of Development Types of work activity being performed at any one time. Are all programmers involved in active development? or are some involved in planning for new projects, some involved in new design, and some involved in design maintenance?
- d. Environment An organization structure must recognize and be able to cope with the corporate environment in which it exists. Two important considerations are: First, authority delegated to the organization and second, interfaces with other organizations.

### Different Structures

Although the organization of software development can follow one of the three standard forms of organization structure (functional, project, matrix) most large departments are too complex to strictly accommodate only one form, thus they combine two or three of the standard organization structures.

I would like to describe my experience with various organization structures and how these structures affect the process of software development.

# Functional Organization

Chart 14 shows the structure of a functional organization which has been used to control the development of a large software controlled system.

# - Disadvantages

- a. Tends to limit the horizontal growth of programmers. As an example - in a true functional organization few support programmers' get involved in application type programming. Also few hardware designers get deeply involved in software development. This structure is definitely not prone to growing generalists.
  Functional groups also have a tendency of promoting their specialities rather than working toward a unified project goal. Project perspective can easily get lost.
- Resolution of "project type" technical and administrative problems are usually made by the only common link in the organization - the director. In many instances, technical problems arise of such detailed nature that the director cannot make a proper analysis; often, the "loudest" or technically strongest side wins - to the detriment of the entire project.
- c. This type of structure is prone to instability. If the director flounders or lacks true leadership, the entire organization has a difficult time progressing. Due to the nature of its structure, the . functional organization cannot tolerate either overly strong managers or an overly weak director.

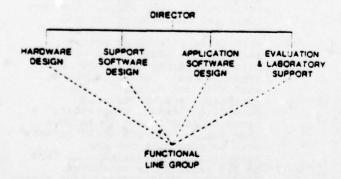
#### - Advantages

- a. For a strong director (hopefully free from megalomania) this organization sets a stage for very tight and centralized control.
- b. Work force efficiency and job work force efficiency and job security are often noted as the major advantages of a functional organization - "where people do what they can do best". These advantages are not totally justified. First, programmers normally do not want to do what they can do best but would rather work on something new and technically challenging. As far as job security is concerned, functional organizations have always been noted for their intrinsic security due to the fact that they are not project oriented, where the loss of project pinpoints a group of programmers as being "out of work". I have never experienced the advantage of security in a

functional organization. The security of a programmer depends on four factors - in any organization structure.

- 1. His work output.
- 2. Conduct.
- His ability to work in more than one discipline.
- 4. Years with company.
- c. Probably the greatest single advantage of a functional organization is that it has the intrinsic ability to constantly improve software development methodologies and development procedures. A soft-ware "load team" will be used as an example. In a functional organization this team will serve many projects. Over a period of years the "load team" members become very efficient. In a project organization (which will be described later) the load team is created only for that phase of a project when "software loads" are necessary (certain software loads are not needed during the specification stage of software development). Thus, the sporadicappearing load team in a project organization neither has time nor sufficient experience to generate good procedures - they seem to always be learning. The load team has been used as an example; however, the same advantage exists in a functional organization for all facets of software development from specification through design maintenance.

# CHART 14 FUNCTIONAL ORGANIZATION



# Project Organization

One must be careful when looking at an organizational structure and assuming that it is project in nature. In many instances a department which is considered to be project organized as shown in Chart 15, is

truly functional in operation - at least at the supervisor level - which most critically determines the working environment for programmers. Chart 15 shows that although the organization is "project" when using the department as a reference, it if "functional" when using the supervisor as a reference point. In truth, the programmer works under the same conditions as if the entire department used a functional structure. The same advantages and disadvantages as listed for a functional organization still exist - but now the unit head replaces the director as the center of all major decisions.

An important conclusion of this analysis is that all structures tend to look project oriented as one uses higher points of reference.

An organization acquires a true project structure only when the programmer sees its effect - and this exists only when a significant portion (70%) of an entire development is under the direction of a first level supervisor.

With this situation the project structure acquires the following advantages and disadvantages:

### - Disadvantages

- a. Projects must be kept small it is very difficult for a supervisor to directly control more than twelve programmers.
- b. Control on a department level is difficult since most decisions are made at low levels.
- c. Critical manpower (specialists in operating systems) is difficult to acquire for each small project.
- d. Training is high and costly.
- Specific design standards and procedures either do not exist or are not uniformly followed.
- f. Full "men" must be made available to each project. Thus, a specialist who, under a functional system, would distribute his talents among many projects, must now be assigned to one project and would have to work in areas which are not his specialty in order to remain fully employed.
- g. Movement of people from one project to a more important project is usually more difficult than in a functional structure.

### - Advantages

a. Decisions are made at the lowest possible level in the organization. Thus allowing quicker decisions and better project control.

- b. Since full responsibility for the project is under the control of one supervisor, interfaces are minimized and work output is very high, decisions are quickly made.
- c. This type of structure tends to change specialists into system generalists since the specialist is forced to work in many areas.

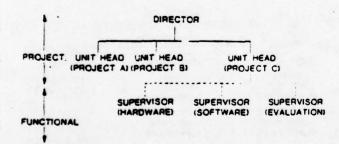
  Management personnel also develop faster in a project organization than in a functional organization.
- d. Motivation is high most programmers would rather work on small, short interval projects, where they can see all aspects of a single project.
- Major interfaces are kept within one group. This improves project communication.

A true project organization works effectively when a department has responsibility for many independent small projects and has matured to a state where supervision is strong and technical talent is abundant.

A major difficulty in directing a large department which is made up of many small projects is to insure a backlog of new projects so that new work can constantly be pushed into line groups as old work is completed.

# CHART 15 PROJECT ORGANIZATION

- FALSE -



## Matrix Organization

In most instances a functional type organization is required to support large projects - at least functional from a programmers' viewpoint.

A cure for some of the problems encountered in a functional organization can be achieved by employing a matrix structure.

Chart 16 shows a diagram of how a matrix structure is put together. The standard way to form a matrix is to first appoint a project manager, give him no line authority, give him all technical project responsibility and then place a few system

analysts under his control.

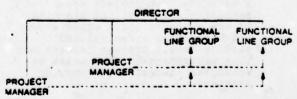
# - Disadvantages

- a. The Project manager has a difficult job. He has the responsibility originally held by the director, but unlike the director the project manager has no real authority. It is possible to try to partially offset this authority problem by telling the line unit heads that the project manager has budget control; but it soon becomes obvious that unit heads must retain responsibility for budgets. The project manager can only complain to the director.
- b. Matrix organization causes two major conflicts: Project manager conflicts with director since both are trying to perform the same job. Project manager conflicts with line unit heads since, in many cases, they are also trying to perform the same job.

### - Advantages

- a. Department has a built-in technical cross-check. It should be noted that a similar type of development cross-check can be achieved by rotating line unit heads or line supervisors (but not both simultaneously) at a time which is early enough to allow for project recovery if a serious design flaw is discovered.
- b. A project manager and his staff are better equipped to make project type technical decisions than either the director or any single line unit head.
- c. The project manager and his team form a good place to assign schedule coordination, specification control, and special project studies. This group can also be assigned initial planning for future new projects.

# CHART 16 MATRIX ORGANIZATION



-PROJECT MANAGER CONTRACTS WORK TO FUNCTIONAL LINE GROUP -ONE PROJECT MANAGER FOR EACH MAJOR PROJECT

### Composite Organization

A flexible organization structure is one which can cope with both large and small software projects; and simultaneously allow each project to rxist at any stage of active development or design maintenance.

A structure which has successfully worked in such an environment exists as a combination of all three standard organization structures. Functional, project, matrix. I have called this structure a composite organization and it is illustrated in Chart 17.

Each of the previous organization structures has certain advantages and disadvantages. Each structure is designed to be effective in a specific work environment. Organizations developing one large software project will tend to be functional; organizations with many small projects will tend to be project. Organizations with a few large projects will tend to be matrix. To properly deal with more complex environments, a composite organization has been established.

Important characteristics of a composite organization are:

- a. In a manner similar to the matrix organization, a project leader is appointed to coordinate project activities and schedules. Unlike the matrix organization, the project leader is not responsible for total project success. He is responsible for uncovering discrepancies and reporting his results to responsible line management.
- b. The composite organization recognizes that close cooperation is required between active design groups and "design maintenance" groups. It places a single "design maintenance" supervisory group under each unit head having new design responsibility. This process allows for efficient movement of program responsibility between supervisors responsible for active design and supervisors responsible for design maintenance.
- c. The composite organization includes a separate "new project group" responsible for initial planning of new projects and a staff group which provides a centralized service to all project leaders and line management in such areas as schedules, budgets and computer services.
- d. The composite organization includes a separate evaluation group. This group is used to provide independent evaluation of all software design. This group also maintains system prototypes and supporting computers. It is responsible for configuration

- control of all software that has been commercially released as well as long term interface with "users" of the program.
- e. The composite organization can act as a project organization by allowing a single supervisor to directly control a complete project. For those projects requiring a more functional arrangement, the composite structure allows a supervisor to directly control only the major portion of a project and contract to the functional part of the composite organization for critical design assistance.
- f. The composite organization can act as a true functional organization for projects requiring many different critical resources.
- g. The composite organization recognizes the need for teams and thereby allows for team leaders (other programmers) to acquire a higher technical status than team members. However, all team leaders and team members report directly to the supervisor for task assignment and other administrative guidance. Team members consist of both programmers and technical assistants. The function of an assistant varies among groups: however, an assistant is usually involved in documentation, unit testing, editing and many software bookkeeping duties.

# CHART 17 COMPOSITE STRUCTURE

NEW PROJECT GROUP -	DIRECTOR	- STAFF GROUP	
UNIT HEAD	UNIT HEAD	UNIT HEAD	
SMALL PROJECT SUPERVISOR	SMACL PROJECT SUPERVISOR	FIELD SUPPORT & SUPERVISOR	
FUNCTIONAL HARDWARE	FUNCTIONAL SOFTWARE	EVALUATION SUPERVISOR	
SUPERVISOR	SUPERVISOR	CONFIGURATION	
DESIGN MTC	DESIGN MTC	CONTROL	
SUPERVISOR -	SUPERVISOR	SUPERVISOR	

PROJECT LEADERS MAY EXIST AT ANY LEVEL IN THE ORGANIZATION: FOR LARGE PROJECTS HE MAY REPORT TO DIRECTOR, FUR SMALL PROJECTS HE MAY REPORT TO SUPERVISOR

## Conclusion

Management of software development is progressing through a very rapid stage of maturity. As management becomes more experienced, complex software projects are being developed on schedule and within budget. This paper has stressed two points. First, software developments should be based on historical experience. Preferably using experience gained within the supporting company. Second, software developments must

have established management objectives, a flexible organization structure and a predefined development methodology if management is going to maximize output with minimum expenditures. Although this paper does not concentrate on the similarities between managing hardware development and software development, experience is showing that the same management principles are applicable to both processes.

The intent of this paper has been to give an overview of a few major areas associated with software management. As with all areas of management there are no answers - only approaches - that work sometimes.

## RESOURCE ESTIMATION

# Jules Schwartz

# Computer Sciences Corporation

# Introduction

The estimation of resources is probably the least precise aspect of system development. On the assumption, however, that an orderly attempt at determining resources is necessary prior to the start of a project, simple guidelines for estimation have been developed.

Although these techniques cannot be relied upon to produce exact results, they serve a number of useful purposes:

- 1. They tend to produce conservative results, which is counter to most people's original, if not final, estimates.
- 2. They can be reexamined as a project proceeds, and better estimates of each factor can be made.
- 3. It makes people think of many important ingredients before plunging ahead.

# SLIDE 1 - FACTORS FOR CONTROL OF ESTIMATES

Ingredients for resource estimating range from clearly quantitative factors for which measurement of effect can be stated reasonably well to subjective factors which require considerable thought before a numerical contribution to the estimate can be made. Slide 1 provides labels for five types of estimation parameters. The first (PROGRAM SIZE) is approximate and can be derived through experience, rough design, prototyping, or the use of consultants. The accuracy of this factor clearly improves as the work progresses.

The other labels represent, from top to bottom, factors for which decreasing levels of confidence in precise numbers are possible. Nevertheless, the estimate must take all of these into account.

# FACTORS FOR CONTROL OF ESTIMATES

- PROGRAM SIZE-ESTIMATE
- STATISTICAL-MEASURED
- OBSERVED-APPROXIMATE
- · OBSERVED-JUDGMENTAL
- OBSERVED-APPLICABLE

## SLIDE 2 - STATISTICAL

The statistical factors are those for which there has been enough experimentation or experience to foster reasonable confidence in their effect on costs.

"A" may be derived for a particular organization over time, utilizing results from a number of projects, or one may utilize "industry averages" where they seem appropriate (e.g., 10 lines per day). "A" in this case covers the time period between the beginning of Detailed Design and the end of Integration.

"C" case refers to the difficulty in programming. At one extreme would be a batch, serial process. At the other end would be a real-time, communications-oriented random process.

"P", "L", and "I" are all reasonably clear.

Thus, using just these factors and System Size(S), the estimate for required resources can be approximated.

# STATISTICAL

- AVERAGE NUMBER OF INSTRUCTIONS PER DAY (A)
- EFFECTS OF COMPLEXITY (C)
- EFFECTS OF PEOPLE (P)
- PROGRAMMING LANGUAGE (L)
- DEBUGGING STYLE (I)
- REQUIRED RESOURCES = F (S,A,C,P,L,I,...)

# SLIDE 3 - ESTIMATE USING STATISTICAL FACTORS

R.R. on this slide represents the calculation when only statistical factors are used for Resources Required. The effect of each factor is listed at the bottom of the slide.

# ESTIMATE USING STATISTICAL FACTORS

S = SIZE IN HIGHER LEVEL SOURCE LINES

A = INSTRUCTIONS/PROGRAMMER/DAY

$$1/2 \le C \le 2$$

$$\frac{1}{3} \le P \le 3$$

$$1 \le L \le 2$$

$$1/2 \le 1 \le 1$$

# SLIDE 4 - APPROXIMATIONS

These are factors (and other known situations) which affect the resources required for a system development.

"ALLOWED TIME" influences a project in the sense that it can actually cost more for periods of time which are unreasonably short. This is illustrated further on Slide 5, and an approximation (M) of its quantitative effect is given on Slide 8.

Experience (X) with similar efforts will normally improve performance considerably, although unless the experience is highly similar it won't be very beneficial.

The effects of other factors will be explained in the following slide.

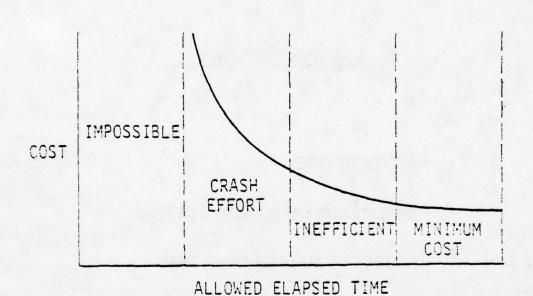
# **APPROXIMATIONS**

- · ALLOWED TIME
- EXPERIENCE ON SIMILAR SYSTEMS
- RESOURCE BUILDUP OVER TIME
- · SUPPORT PERSONNEL

# SLIDE 5 - TIME RANGES

This is one author's depiction of the beneficial effect of allowing more time on a project. In addition, it is shown that it is impossible to complete some projects within certain time periods, no matter how much money is spent. This is clearly an approximate curve, and the actual factor used in the calculations (M) provides a linear estimate.

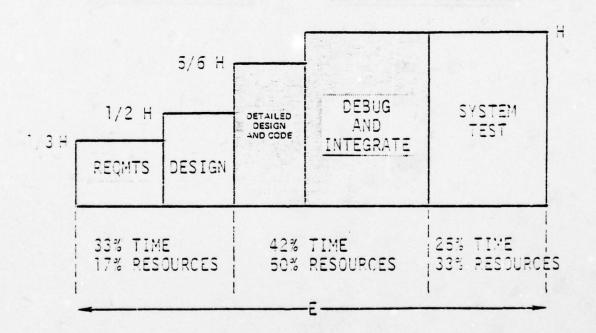
# TIME RANGES



# SLIDE 6 - TECHNICAL MANPOWER RESOURCES OVER TIME

This slide depicts "typical" growth rates in necessary technical manpower over the lifetime of a system development. "H" is the lifetime of a system development. "H" is the maximum number of people utilized and "E" is total elapsed time from the beginning of Requirements Specification to the end of System Test. The heavily shaded area represents the period during which the factor "A" is computed for this technique. Along the abcissa, "typical" percentages of time and personnel are shown for different phases of a project. These (and the shape of the curve) will vary widely as a function of the parameters utilized for estimating, but this chart represents a reasonable starting point. This chart doesn't reflect the overlapped nature of most projects, in the sense that work doesn't stop for each phase as neatly as depicted here. The growth isn't normally as much a discontinous step-function as shown here.

# TECHNICAL MANPOWER RESOURCES OVER TIME



# SLIDE 7 - TOTAL RESOURCES

This slide shows the need for personnel in addition to the technical implementors discussed earlier. Included are managers, liaison personnel, staff members, documentors, and operations personnel. The major point of the slide is to illustrate the way in which the percentage of "overhead" personnel increases as the size of the technical staff grows. Of course, this factor is subject to variation with different organizations.

# TOTAL RESOURCES

TECHNICAL PERSONNEL	FACTOR REQUIRED SUPPORT PERSONNEL	
<12	.5	
12 — 100	1	
>100	>1	

# SLIDE 8 - ESTIMATE ADDING APPROXIMATE FACTORS

The expression for Technical Manpower Required Resources (R.R.) is given in (1). Included also are the expression (M) which takes elapsed time into account and the limits on the factor for experience (X). This expression includes only the Statistical and Approximate Factors.

The expression in (2) gives the calculation for the size of technical resources during System Test in terms of the expression computed in (1).

Expressions (3) and (4) give calculations for Requirements and General Design Specification Development.

# ESTIMATE ADDING APPROXIMATE FACTORS

(1) R.R. = 
$$\frac{S}{A} + C + P + L + I + M + X ...$$
  
WHERE: M =  $\frac{2 R - E}{R}$ 

$$1/5 \le X \le 1$$

(2) R.R. (SYST.TEST) = 
$$\frac{2}{3}$$
 R.R.

# SLIDE 9 - JUDGMENTAL FACTORS

Some of the major potential problem areas which may arise on a given project are depicted in Slide 9. In the main they point out the costs that should be associated when plans are made dependent on unknowns or clearly weak technical assumptions, including hopes for acquiring people from other sources, a willingness or need to change specifications during development, weak managers, outstandingly difficult technical problems, and reliance on untried hardware or software. The precise amount to add to the cost of a project for these situations is difficult to specify in general, but increases in cost from 10 to 30 percent should not be unusual.

# JUDGMENTAL FACTORS

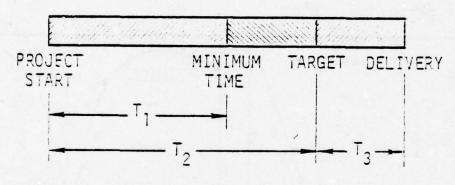
- RELIANCE ON PERSONNEL DEPARTMENT
- DEPENDENCE ON OTHER PROJECTS
- TOLERANCE FOR CHANGE
- EXPERIENCED FAILURE
- BREAKING TECHNOLOGICAL BARRIERS
- · TRUST IN THE SUPPLIER
- · OTHERS

INCREASE NEEDED RESOURCES APPROPRIATELY, THEN, USE SUPPORT FACTOR

# SLIDE 10 - SIMPLE OBSERVATIONS

This slide illustrates some unverified conclusions about the conduct of many system development efforts. The main points are that most projects slip to some extent over the time originally planned, and the amount slipped (T3 on the Slide) is almost independent of the time originally scheduled.

# SIMPLE OBSERVATIONS



 $T_3$  ALMOST INDEPENDENT OF  $T_2$  IF  $T_2 > T_1$ OBJECTIVE: MINIMIZE  $T_3$ 

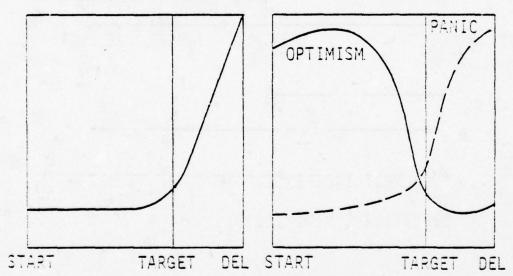
# SLIDE 11 - MORE OBSERVATIONS

This Slide tries to illustrate why the observations of Slide 9 may be true.

Basically they occur because of human nature and managerial decisions. If these curves are anywhere near accurate, it will be seen that the bulk of the work doesn't get done until near (and sometimes after) the target date. This is probably largely due to the fact that many plans don't include much in the way of tangible output until the target date.

# MORE OBSERVATIONS

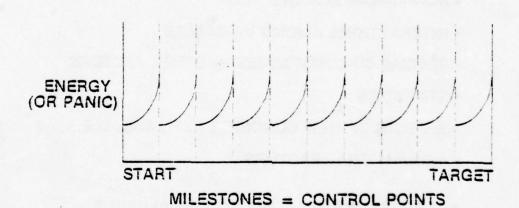




# SLIDE 12 - OBJECTIVE: MANY MILESTONES

This slide points out the two valuable features of providing many tangible deliveries over the lifetime of a system development. The first advantage is that the work force is always motivated to produce results and thus achieve at a higher rate throughout the life of the project. The second advantage is that each tangible deliverable milestone provides an excellent window on progress.

# **OBJECTIVE: MANY MILESTONES**



129

## SLIDE 13 - SUMMARY OF PROBLEM

This slide lists some of the difficulties in utilizing the preceding techniques. It is clear that even when estimating techniques are followed, estimates are still only rough approximations. However, this does not imply that simple guesswork is better. Disciplined attempts at recognizing and appreciating the influence of all possible factors is essential if a good approximation is to be achieved.

# SUMMARY OF PROBLEM

- ESTIMATES OF SIZE AND COMPLEXITY
- NON-LINEAR EFFECTS
- INTERACTIONS AMONG VARIABLES
- SPECIAL CIRCUMSTANCES OTHER FACTORS
- STATISTICS
- EFFECTS OF NEW CONCEPTS (D.P., S.P., D.B. . . .)
- MOTIVATIONAL FACTORS

BUT DISCIPLINE IN ESTIMATION IS VALUABLE.

# The Cost of Developing Large-Scale Software\*

RAY W. WOLVERTON, MEMBER, IEEE

Abstract-The work of software cost forecasting falls into two parts. First we make what we call structural forecasts, and then we calculate the absolute dollar-volume forecasts. Structural forecasts describe the technology and function of a software project, but not its size. We allocate resources (costs) over the project's life cycle from the structural forecasts. Judgment, technical knowledge, and econometric research should combine in making the structural forecasts. A methodology based on a 25 imes 7 structural forecast matrix that has been used by TRW with good results over the past few years is presented in this paper. With the structural forecast in hand, we go on to calculate the absolute dollar-volume forecasts. The general logic followed in "absolute" cost estimating can be based on either a mental process or an explicit algorithm. A cost estimating algorithm is presented and five tradition methods of software cost forecasting are described: top-down estimating, similarities and differences estimating, ratio estimating, standards estimating, and bottom-up estimating. All forecasting methods suffer from the need for a valid cost data base for many estimating situations. Software information elements that experience has shown to be useful in establishing such a data base are given in the body of the paper. Major pricing pitfalls are identified. Two case studies are presented that illustrate the software cost forecasting methodology and historical results. Topics for further work and study are suggested.

Index Terms—Computer programmer code productivity rates, cost data base used in cost estimation mechanism, cost of developing custom software, incentive fee structure influence on computer software development, principles of pricing computer software proposals, resource allocation in computer program development, software cost estimating methods and general logic, software cost estimation algorithm, software development and test life cycle—cost impact, systems approach to pricing large-scale software projects.

## INTRODUCTION

ESTIMATING the cost of a large-scale computer program has traditionally been a risky undertaking. Now that software has been with us for nearly two decades, it is reasonable to hope that we have learned something that would make our predictions less unreliable. At TRW we have been motivated to make a serious effort toward improving our software estimating methods for large-scale software systems, for a set of very compelling reasons.

 Our customers have shown a growing unwillingness to accept cost and schedule overruns unless the penalties were increasingly borne by the software developer.

 Partly as a consequence of 1), we have entered into software development contracts where both adherence to predicted costs and on-time delivery were incentivized. That means we made more money if we could predict accurately the cost and the time it would take to do the job.

3) We found that we could improve our estimates only by improving our understanding of exactly what steps and processes were involved in software development, and this understanding enabled us to manage the effort better. The better management, in turn, improved our estimates.

This paper presents the essential results of our efforts to improve our software cost estimating techniques. It should be understood that the specific contracts that are used here were government contracts whose particular provisions shaped those efforts to some degree. Nevertheless, we feel that much of what we learned is of general interest and can be applied or adapted to nearly any software development program of any size.

We can start with a review of some of the ways in which software development differs from hardware development, and which have traditionally contributed to the problem of estimating software costs. One of these ways is the problem of managing the people. The nature of programmers is such that interesting work gets done at the expense of dull work, and documentation is dull work. Doing the job in a clever way tends to be a more important consideration than getting it done adequately, on time, and at reasonable cost. Programmers tend to be optimistic, not realistic, and their time estimates for task completion reflect this tendency. The software engineer, the mathematician, and the programmer have trouble communicating. Visible signs of programming progress are almost totally lacking.

Another set of difficulties arises from the nature of the product. There are virtually no objective standards or measures by which to evaluate the progress of computer program development. Software characteristics are often in conflict, requiring frequent tradeoffs among such factors as core storage requirement versus tight code, fast execution time, external storage to minimize I/O time, maintainability by the eventual user, flexibility or adaptability to new needs, accuracy and reliability, and self-check and fail-safe modes of operation. The applications software developer finds himself in a dynamically evolving and constantly changing environment.

Software planning has problems stemming from the natural desire to get going, which means taking shortcuts. What are the steps that should precede the start of coding, and how does the knowledgeable manager allocate his limited resources? Planning also is made difficult by the

The author is with the TRW Systems Group, Redondo Beach, Calif. 90278.

<sup>\*</sup>Previously published in <a>IEEE Transactions on Computers</a>, Vol. 23, No. 6, 1974.

problem of translating the customer's functional requirements into software requirements, and later defining the procedures by which you determine that those original customer requirements have been met. It is not always easy to determine when you are through and whether you have delivered what you said you were going to deliver.

The overall approach we have developed for dealing with these problems and producing reasonably reliable cost estimates for large-scale software systems consists of the following basic elements, which will be discussed in detail in the remainder of this paper.

- Understanding exactly what the software development process consists of over its life cycle.
  - 2) Recognizing the pitfalls in the pricing of software.
- Establishing and maintaining a good data base reflecting our history of actual software development costs.
- 4) Determining the most cost-effective allocation of resources over the entire period of the expected contract.

#### THE SOFTWARE DEVELOPMENT CYCLE

Because software is during most of its development cycle (and even afterward) a basically intangible product, it is even harder to control than a complex hardware system. We have learned by trial and error that no cost predictions can be fulfilled unless the mechanism for management control is solved in advance. We have established five management principles that apply throughout the software development cycle to reduce the problem of control to manageable size.

- 1) Produce software documentation that meets established programming standards, is accurate, is produced according to the needs of the audience who will use it, and, is most of all, an instrument by which management controls the project.
- 2) Conduct technical reviews against predetermined acceptance criteria to the end that, upon customer approval, predetermined baselines are established.
- 3) Control software physical media (tapes, disks, decks) to assure use of a known configuration in testing, demonstration, certification, and delivery.
- 4) Apply software configuration management controls and procedures to assure that required changes to baselines are implemented, tested, fully documented, and understood.
- 5) Provide a data reporting, repository, and control system to assure that all problems, deficiencies, change proposals, and software configuration data are analyzed, reported, recoverable, and available to all project and user personnel when and where needed.

Technical reviews are required during the software development life cycle. Reviews ensure that the resulting software products meet the requirements established for that level of completeness and understanding. A "baseline" is established at designated points in time during the software development cycle when the requirements (or

design) have been defined, documented, reviewed, and approved. From that time forward, that baseline serves as the reference point for evaluating and managing any changes in scope, price, or schedule.

In the specific cases to be discussed, certain of these principles were incorporated in government software procurement policies applied to the contract, but no manager of a large software development should ignore them whether they are required by his customer or not. For example, without software configuration management—configuration identification, configuration control, configuration status accounting, and verification—the software development manager will spend much of his time solving problems that need not have arisen. Software is controlled through control of documentation within a structured software development process.

There are several ways of analyzing the software development process, but the one we have chosen leads to the definition of seven phases, or steps, each ending in a discrete event or document (see Fig. 1). The steps are basically sequential, but in practice there are iterations between adjacent steps and even between any of the steps. The seven steps are as follows.

Step 1—Performance and Design Requirements: This document establishes the requirements for performance, design, test, and qualification of the software at the system level

Step 2—Implementation Concept and Test Plan: This document provides, at an early date, the preliminary design concepts that meet the requirements, and identifies the preferred approaches, design tradeoffs, and alternatives, and leaves no high-risk technology area unresolved.

Step 3—Interface and Data Requirements Specification: This document defines the interfaces between subsystems and major elements within subsystems, including presentation of table and file structure, data origin, destination, and update characteristics.

Step 4—Detailed Design Specification: This document provides, in precise detail, the complete design and acceptance specifications for the software at the routine level, which includes detailed flow charts, equations, and logic. It is the source material from which the program is coded.

Step 5—Coding and Debugging: Upon design review and approval of the Step 4 documentation package, which may be staggered in time and organized into volumes dealing with a single computer program module, the coding of the software can begin. This step is devoted to coding and debugging (and amending documentation), using debugging aids and desk check of the first compilations. The phase terminates for a given module when it has passed the development-level verification tests imposed by the designer, and he turns the module over for assembly onto the master tape for the beginning of independent system validation testing.

Step 6-System Validation Testing: As the software

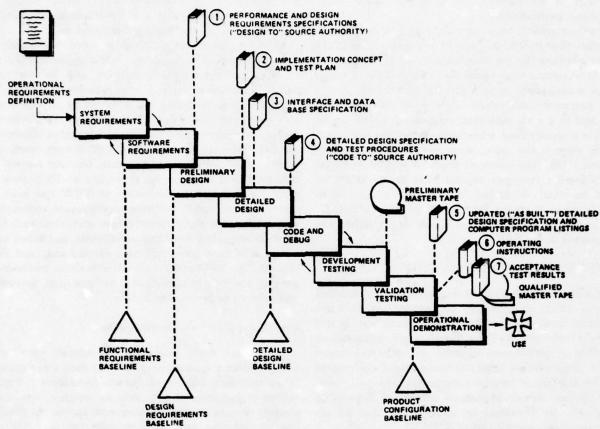


Fig. 1. Typical custom software development and test steps, showing seven unique documents.

modules are released by the design group for assembly onto the master tape, the buildup of the software package into successively more complex interfaces and capabilities can commence. When two or more modules are assembled, the first stages of "system testing" can get underway according to the test plan started at Step 2 and upgraded in level of detail at Step 4 to include test procedures and quantitative acceptance criteria. The phase terminates when all modules have been successfully tested against the predetermined acceptance criteria at the system level.

Step 7—Certification and Acceptance Demonstration: This final step is devoted to formal acceptance of the software system by subjecting it to previously defined acceptance and test specification procedures, which are executed in as near an operational environment and host computer configuration as possible. The phase terminates with successful "operational demonstration," witnessed by the customer and his quality assurance team. The software packages are then ready for installation, integration, and checkout in the operational environment.

The final step in the software development life cycle is the forerunner of the operations and maintenance phase, strictly speaking, Step 8. This phase is concerned with installation, integration, and checkout in the full operational configuration. The software contractor assists in any way required to integrate the software products of the development, or acquisition, phase into an operational software package that meets the original (and updated) performance and design requirements, the product of Step 1 and subsequent updates. The activities of this phase will also include training, rehearsal support, software problem reporting, fault isolation and correction, formal problem closure and documentation update, and finally, first mission operations support. Because this is not part of the software development effort, we cost it differently—usually as a level of effort.

#### PRICING PITFALLS

Pricing is the complete process by which we arrive at a price that we quote to the customer. Cost estimating is a major part of that process, although it frequently has to be done several times before a final price is arrived at. The general principles involved in pricing large R&D efforts of any kind have been defined by Beveridge [1], and apply to large software developments as well. Beveridge's discussion centers around the preparation of a proposal in response to a government Request for Proposal (RFP), but his common-sense principles are

applicable to any complex pricing exercise where many individuals are involved.

There is a general tendency on the part of designers to gold-plate their individual parts of any system, but in the case of software the tendency is both stronger and more difficult to control than in the case of hardware. A major task of management is to make certain that the design being proposed (and priced) does meet the customer's need, and that each individual component design can be traced to a specific need, while at the same time it does not provide more (more speed, more accuracy, less need for core, etc.) than the customer needs or wants. Anything being offered a customer beyond what he has asked for should be clearly identified (and priced) as an option. That is what is meant by the "fixed-price attitude": what is the absolute minimum I can do to satisfy the needs stated in the RFP?

### SOFTWARE COST ESTIMATING METHODS

The software industry is young, growing, and marked by rapid change in technology and application. It is not surprising, then, that the ability to estimate costs is still relatively undeveloped. Even though many organizations are trying to devise more scientific and objective means for estimating costs of large software systems, the present state of the art is largely judgmental. We will, however, discuss certain important exceptions shortly. A review of other firms and agencies confirm that we are facing a problem common to all [2]-[5].

Estimating the cost of producing computer software relies heavily on the judgment of experienced performers. The software analyst, or estimator, normally breaks the total job into elements that are estimated separately and then summarized into an estimate for the total job. The estimating analysis and synthesis may appear as a mental process or may involve an explicit algorithm [6].

In either case, an empirical data base is used as an objective reference, and the estimator uses his judgment to account for differences. Pieces of information used in the comparisons and adjustments for differences include: a) analysis of initial requirements; b) allocation of requirements to software modules; c) estimates of number of object instructions per module; d) complexity and technological risk; e) user environment and characteristics of the customer; f) computer of choice and interchangeability among other user sites; g) higher order language of choice or criteria for use of assembly language; h) type of software to be developed; i) whether software is to be delivered to an operational user; j) technical experience on that type of job; k) capabilities of the member of the technical staff who probably will do the work; l) type of fee and its incentive structure; m) length of development time; n) single-model multiple-release versus multiplemodel single-release development concept; o) performance record of other large-scale systems (AWAC, SAGE, etc.) in the number of instructions and development manmonths; and p) management factors to do with productivity rates, error rates, work environment, availability of computer time, and many other variables.

Most estimators use a logical sequence in establishing their estimates. The logic uses exchange coefficients between some measurable parameter and its cost and adjustment factors derived from experience. One of the pitfalls is that estimating ratios should be directly traceable to recorded cost facts (not hearsay), and even the factual data can contain varying and unstated allowances for risk (we may have assumed a 40-h work week, our records show we charged that rate, but our records do not show that most personnel worked 45-50 h/week or more). Our long-term objective at TRW has been to create a systematic software development estimation system that can significantly reduce statistical variances between estimated costs and actual costs, and is not only accurate in that sense but also simple, fast, and convenient. We will briefly look at estimation methods in general, and two in particular that have been developed and reduced to practice at TRW.

# General Logic Followed in Estimating

Traditional cost estimating procedures start with fixing the size of each activity, its start date, and duration. When necessary, adjustments are made to account for the caliber of performer personnel to be assigned, risk, complexity, uncertainties in requirements, and so on. Finally, the amount and type of manpower (man-months per month) and computing resources (hours per month) are converted to dollar costs by applying bid rates. Other direct charges (documentation costs, travel, etc.) are added, and summaries are made through the pricing system. Traditional methods can be classified as one or more of the techniques described below.

- 1) Top-Down Estimating: The estimator relies on the total cost or the cost of large portions of previous projects that have been completed to estimate the cost of all or large portions of the project to be estimated. History coupled with informed opinion (or intuition) is used to allocate costs between packages. Among its many pitfalls is the substantial risk of overlooking special or difficult technical problems that may be buried in the project tasks, and the lack of details needed for cost justification.
- 2) Similarities and Differences Estimating: The estimator breaks down the jobs to be accomplished to a level of detail where the similarities to and differences from previous projects are most evident. Work units that cannot be compared are estimated separately by some other method.
- 3) Ratio Estimating: The estimator relies on sensitivity coefficients or exchange ratios that are invariant (within limits) to the details of design. The software analyst estimates the size of a module by its number of object instructions, classifies it by type, and evaluates its relative complexity. An appropriate cost matrix is constructed

from a cost data base in terms of cost per instruction, for that type of software, at that relative complexity level. Other ratios, empirically derived, can be used in the total estimation process, for instance, computer usage rate based on central processing unit (CPU) time per instruction, peripheral usage to CPU usage, engineers per secretary, and so forth. The method is simple, fast, convenient, and useful in the proposal environment and beyond. It suffers, as do all methods, from the need for a valid cost data base for many estimating situations (business versus scientific, real-time versus nonreal-time, operational versus nonoperational).

4) Standards Estimating: The estimator relies on standards of performance that have been systematically developed. These standards then become stable reference points from which new tasks can be calibrated. Many mature industries, such as manufacturing and construction, use this method routinely. The method is accurate only when the same operations have been performed repeatedly and good records are available. The pitfall is that custom software development is not "performed repeatedly."

5) Bottom-Up Estimating: This is the technique most commonly used in estimating government research and development contracts. The total job is broken down into relatively small work packages and work units. The work breakdown is continued until it is reasonably clear what steps and talents are involved in doing each task. Each task is then estimated and the costs are pyramided to form the total project cost. An advantage of this technique is that the job of estimating can be distributed to the people who will do the work. A difficulty is the lack of immediate perspective of the most important parameter of all: the total cost of the project. In doing detailed estimates, the estimator is not sensitive to the reasonableness of the total cost of the software package. Therefore, top-down estimation is used as a check on the bottom-up method.

The estimation process in nearly always a combination of two or more of the basic classifications given above. We will briefly discuss two methods that have been used in estimating the cost of developing large-scale software at TRW: a method we used in June 1969 which we will call "smoothing and extrapolation," and the current method we used beginning in October 1970, which we will call "a software cost estimation algorithm."

#### A Systems Approach to Software Cost Estimation

To be as free as possible to deal with the cost estimating method, we will use hypothetical or normalized numbers where numbers are necessary. Assume the software development and test life cycle consists of the seven steps previously defined, except that the starting point at contract go-ahead is immediately on completion of Step 2. We need to understand this assumption clearly, since it establishes the remaining-milestone events which are in-

cluded in the cost which was negotiated with the customer. In other words, the proposal was at a level of detail such that it was, by itself, the implementation concept and test plan. Step 2. The RFP package, together with the bidder's briefing, was very detailed, technically complete, and was. by itself, the performance and design requirements. Step 1.

Our reference project for the most recent past successful software project was identified, and its case history was laid out in terms of the actual events, costs, start dates, and durations. The normalized contract cost as a function of the normalized period of performance is shown in Fig. 2. We will call this case history A. The cost distribution by development activity, that is, the actions that characterize the intervals between discrete milestone events, is shown in Fig. 3. Using case history A we can identify similarities and differences with respect to our cost estimating for our new project. The basic cost method, smoothing and extrapolation, is a combination of earlier methods plus some new ideas for the new project. We will call the new project case history B.

A major development cycle difference was that case A has a baseline design specification deliverable under contract, which accounted for about 12 percent of the total cost as shown by "analysis," and also had the implementation concept and test plan deliverable under contract, which accounted for another 18 percent as shown by "design" (see Fig. 3). Even though the software development cycle was different, we had exact data on the manpower assigned to the balance of the milestone events, which are similar. We now are in a position to extrapolate from case A to case B. Furthermore, since case A had been produced on time and under cost, we were in a strong position for proving demonstrated cost performance on a previous project.

A major functional difference was that case B represented a unified communication, command, and control software system, whereas case A did not have the command function. Immediately, greater attention was given to the analysis and design of that new technology area and its interfaces. Even here, important similarities were identified in the history file because in case A the "similar" command function was the responsibility of another associate contractor. We concentrated on improving the information transfer by having full control of the interface, and by approaching the solution from an entirely new direction. We decided the technical risk was somewhat higher for case B because our cost data base was incomplete, and because the command algorithm design, coding, and testing were innovative. We developed the overall system block diagram, and allocated statement of work tasks to software functions. We were in a position for a first rough cut target cost.

The whole idea in this case was a) to price the total software system from the top down using the experience of the system concept group, b) to price each work

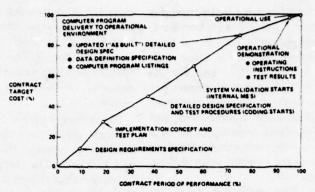


Fig. 2. Typical software development cost experience (case history A).

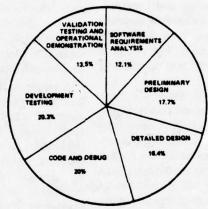


Fig. 3. Cost distribution by activity during full period of performance (case history A). All activities include documentation and travel costs.

package from the bottom up using the experience of each of the five work package managers, and c) to conduct a mock negotiation between the system concept group and each work package group taken one at a time. Differences were derived, costs were traced back to source authority requirements in the customer's statement of work, and adjustments made between the cost, probable risk, and scope of work.

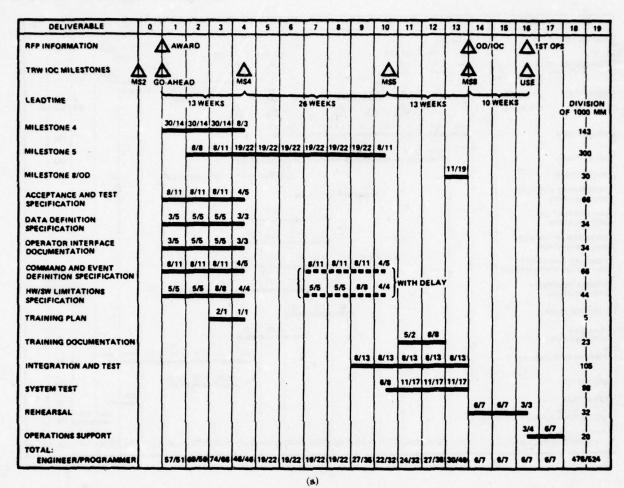
In both methods the basic unit of estimation was the "man-month" for a given task or element. The distribution of the work (man-months per month) was driven by two critical variables: initial operational capability (IOC) and full operational capability (FOC) software configurations were required by the RFP, and a new incentive fee structure was required by the RFP. Each will be briefly addressed by assessing its impact on the cost method.

The distribution, or manpower spread over the proposed period of performance, was initially determined by the system concept group using one new idea. That idea was first to assume that the total development manpower resource was 1000 man-months, and then to determine the shape of the manpower distribution over time that maximized the fee incentive with least risk, such that the

area under the curve remained constant (namely, 1000 man-months). This curve would serve to generate the master schedule for IOC and FOC events. The amplitude of the curve would be the primary focus in the separate negotiations with the five work package managers.

The amplitude represented the number of man-months per month for any given time slice. The sum of the manpower estimates of each of the five work packages at any time point determined that month's manpower requirements, and the sum of those over all time determined the total IOC/FOC manpower requirements, hence total cost. Of the five initial estimates, four were judged too high in the internal first cut negotiations, and one was judged too low.

The shape of the first and second estimate by the system concept group is given in Fig. 4. It is a poor distribution with steep manning rates, high peaks, and sharp declines. Considerable smoothing was required to achieve a unimodal increasing (and decreasing) shape, at comfortable manning rates and reasonable levels and still meet the original goal we set with respect to incentive fee. The shape was drastically altered by introducing the concept of "staggered milestone events." That is, instead



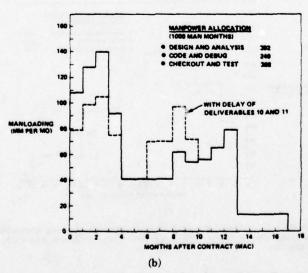
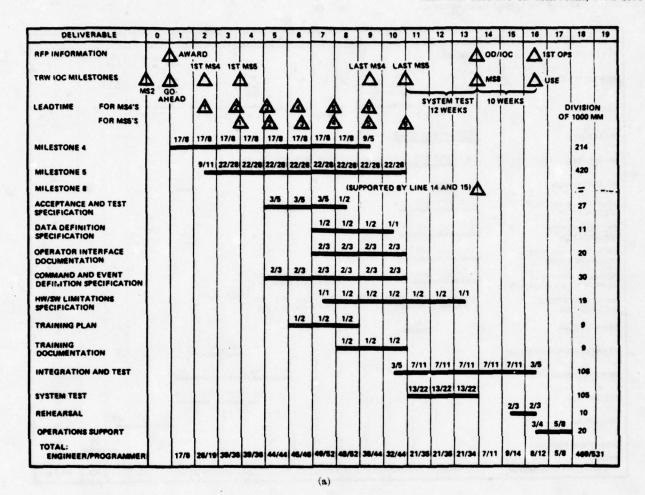


Fig. 4. (a) Manpower estimate with single-release milestone approach, showing key deliverables and labor allocation of 1000 man-months. (b) Manpower estimate with single-release milestone approach, showing uneven labor spread of 1000 man-months.

of one event 4) and one event 5) (steps as previously defined), we planned for a total of six each. We could do this because of modular construction of the software to the end that each module was ready when needed, and

system level testing by the independent test and validation group could commence soon enough to meet all incentives, at least by plan. The smoothed shape of the third estimate, and the one that influenced the time



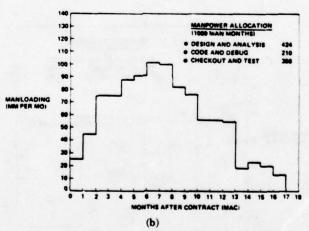


Fig. 5. (a) Manpower estimate with staggered-release milestone approach, showing incremental delivery of detailed design specifications. (b) Manpower estimate with staggered-release milestone approach, showing "smoothed" labor spread of 1000 man-months.

placement of the incentive events, is shown in Fig. 5. Both sets of curves show only IOC activities for simplicity; as IOC man loading phased down, FOC phased up, in general.

A simplified description of the fee structure that influenced the shape of Fig. 5 is that cost, schedule, and per-

formance were incentivized (from a maximum of 15 percent of final target cost to a minimum of 0) based on certain predetermined rules that would be applied at three discrete milestones: Step 4), the detailed design specification, Step 5), the updated detailed design specification with computer program listing, and Step 7), opera-

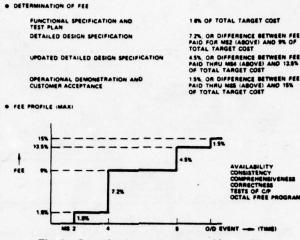


Fig. 6. Incentive fee structure (case history B).

tional demonstration (OD). The plan called for three IOC and three FOC dates.

The effect of a two-step procurement, such as an IOC followed by a later FOC, is as follows [7]. The IOC milestone events and OD constitute the only way that any fee can be earned, in discrete steps as shown in Fig. 6. The earned fee percentages apply to the target cost for the total IOC/FOC task. The incentive fee structure provides for fee penalties for failure to meet the contract dates for satisfactory demonstration of milestone steps 4), 5), and the OD. These penalties are assessed as shown in Fig. 7, on a linearly graduated basis in units of whole days late, where "days late" mean calendar days from the date specified in the contract. The penalties apply separately to both IOC and FOC. That is, the maximum penalty for lateness in demonstrating the OD of the IOC is 15 percent; it is also 15 percent for failure to demonstrate the OD of the FOC. This insures priority to the IOC, but also insures responsible attention to the FOC.

To be effective, the basic incentive structure must be simple, even though it is necessary in the contract to address the major contingencies and allowable options. If the basic incentive structure is not simple it will not readily be grasped by the many people at all levels of the contractor's plant whose work affects the chance of success. If they do not understand it, they will not do anything differently because of the incentive structure. If they do not, the incentive contract will have failed to achieve its fundamental purpose: the people who work on all aspects of the entire undertaking must be conscious of the incentive and must do their work with more care and quality because of it.

The entire incentive approach presumes that the contractor will take specific internal implementing action. It should include some tangible internal management actions that place an additional incentive on the work quality. A clear explanation of the essential features of the incentive structure should be given to all who work in

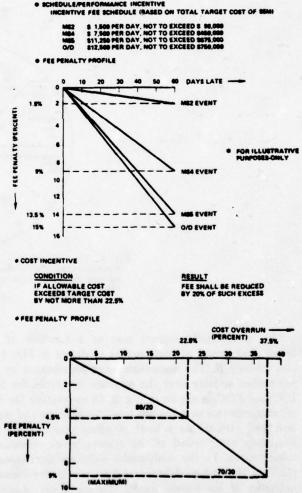
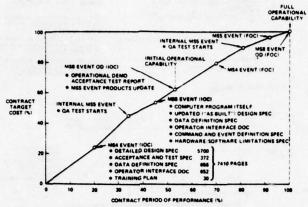


Fig. 7. Penalty assessment structure (case history B).

any manner on the software, keyed to the contribution each one can ake to affect the fee that can be realized by his company.



Typical software development cost experience (case history B-multimodel development).

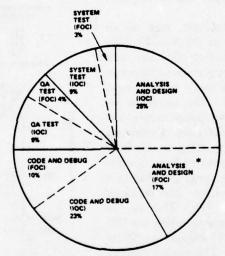


Fig. 9. Cost distribution by activity during full period of performance (case history B—multimodel development). All activities include documentation and travel costs. Distribution excludes O&M costs.

The normalized contract cost as a function of the normalized period of performance is shown in Fig. 8 for case history B. The normalized cost distribution by development activity over the software life cycle, for both IOC and FOC, is shown in Fig. 9. In estimating the cost of large software systems, we treat "operations and maintenance" (O&M) as a level of effort commencing immediately after selloff of the system in its operational environment. In the multimodel software development, O&M of the first model overlaps the purely development activities of the second model. To avoid any misinterpretation of the development costs, the O&M amounted to 15 percent of the total contract value. Comparison of case history A and B in size and cost (man-months), excluding O&M is given below:

	Number of Object Instructions	Man-Months	Rate
Case History A	(I) 102 400	(M) 570	(I/M) 180
Case History B IOC FOC	160 000° 215 000°	945 630 <sup>6</sup>	170 340

\* IOC activities included design and production of a prototype software package of an additional 20 000-object instructions. A slightly modified version was made during FOC.

b Learning curve phenomenon has been verified by empirical data and controlled tests in other industries [8], but not systematically for the software industry.

## A Software Cost Estimation Algorithm

TRW has developed a cost estimation algorithm based on the assumption that costs vary proportionally with the

number of instructions. For each identified routine, the procedure combines an estimate of the number of object instructions, category, relative degree of difficulty, and historic data in dollars per instruction from the cost data base to give a trial estimate of the total cost. The design group estimates the first three software parameters. The system concept group provides the appropriate cost data base used by all designers, as well as the allocation of resources to each phase of the software development cycle, the schedule for each milestone event, and the labor mix. The procedure then spreads the total cost over the period of performance according to these input parameters, gives the man-months per month by labor category, the cost associated with each development activity, and the computer usage by month for checkout and test activities. The output may be considered a "trial estimate" in the sense that the proposal team must be satisfied with the resulting estimate when tested against any conventional method of cost estimating.

The first step in the method is to categorize the software routines that are being considered in the preliminary design. The software categories have been selected based on experience, and are those functionally different kinds of software entities for which a significant cost per instruction (CPI) is expected. Categories that have stood the test of usage in several proposal and preliminary design activities are: a) control routine, which controls execution flow and is nontime-critical, C; b) input/output routine, which transfers data into or out of the computer, I; e) pre- or postalgorithm processor, which manipulates data for subsequent processing or output, P; d) algorithm, which performs logical or mathematical operations. A: e) data management routine, which manages data transfer within the computer, D; and f) time-critical processor, which is a highly optimized machine dependent code. T.

The next step is size and complexity estimates by routine, or subprogram, by the designer. To balance cost and risk, the designer may decide to use software elements that are available to him from a software library and need only some degree of modification or adaptation. At the other extreme, a new technique may be required and be estimated as a high technological risk. To account for the degree of difficulty of a given kind of routine, the designer estimates a risk or complexity factor. This is the most crucial step in the estimating process, for it establishes the cost of the routine with all direct and indirect charges amortized against it. The other steps basically determine how the total cost will be spread over the development cycle.

Two consultants have different views of how software parameters should be estimated, in general. Brandon says a single individual should establish a complexity rating scale (A,B,C,D,E,F) and make a "standard estimate" for each job based on: a) complexity rating of each job;

b) machine used; c) language used; and d) estimated number of instructions. Brandon uses equations fitted to historical data to get standards, and then measures performance against the standard [9].

Lecht believes the estimator should interview the member of the technical staff who will do the job, and negotiate personal agreement on effort. Historical cost data reinforce the estimator's judgment where similar jobs can be found. The estimate is based on: a) similarity with previous modules; b) person doing the job; c) machine used; d) language used; and e) estimated number of instructions. Lecht does not believe meaningful performance standards can be set for software [10].

The simplest technique for narrowing down the many subjective choices early in the estimation process is to ask is the routine new or old, and is it easy, medium, or hard? A complexity rating coefficient can be applied continuously from 1 to 20 as a multiplier, if preferred. The important consideration is allowing sufficient degree of freedom to accommodate a learning experience and individual differences in performance in the effort to obtain a realistic cost. For our present purposes of early preliminary design, we will allow the designer four choices to account for six levels of difficulty for each routine. They are:

	Easy	Medium	Hard	
Old New	OE NE	OM NM	OH NH	

The only parameter that changes as a function of degree of difficulty (OE through NH) is cost per instruction. Override control is available to the designer. Sample data sets will be given in the section on Software Cost Data Base.

The next step is to identify the various development and test phases for producing the software from the conceptual stage to delivery of the operational software to the ultimate user. For our present purposes, the seven steps given in the section entitled The Software Development Cycle will be adopted, but they could be any other steps tailored to the needs of the particular application. For each phase an estimate is required for the fraction of the total amount to be allocated to it. In the case of manpower allocation for distribution of 1000 man-months over the development cycle, it was functionally allocated as 424 man-months to design and analysis, 210 man-months to code and debug, and 366 man-months to check out and test (see Fig. 5). This distribution of 42-21-37 percent at this level of detail has been borne out in practice by several other researchers, as will be demonstrated shortly.

The fourth step is to define the activities in each development phase by means of an activity array and associated cost matrix. The activities as a function of development phase is a  $25 \times 7$  matrix; that is, 25 activities are identified for each of 7 phases. In turn, subsets of the

activities may be summed to "super activity" levels, and the makeup may vary from phase to phase. A typical activity array and cost matrix will be presented in the section entitled Resource Allocation.

The final step in setting up the initial conditions for the cost estimation algorithm is to provide schedule data based on the customer's statement of work, or other management considerations. Schedule data are input as months from go-ahead for each of the milestone periods. Burden rates are input for projected overhead rates, general and administrative, and so forth. Labor mix is defined and unburdened bid rates for the labor grades desired for the phase are defined. Selective cost cuts are under management control by the override capability. Other direct charges are input, which for software is typically travel as a percentage of direct labor costs, such as 3 percent and documentation at 10 percent.

Computer usage data for a machine in the CDC 6500 class with time-shared central processor unit based on sampled data from a programming department by month have been [11]

Number of MTS	Total Number of Processor Units Used	Number of Processor Hours per Man-Month	Peripheral Hours per Man-Month
36	55.6	2.20	7.6
36 35	31.0	1.27	4.4
33	30.0	1.30	4.5
33	46.5	2.02	7.0

Allowing for slightly less power in a 370/155 computer, for instance, a figure of 3 h/month/programmer manmonth would be reasonable. At an average productivity of 1 object instruction/h, this is equivalent to 156 instructions/man-month, or 1.2 min/instruction. The data in the third column are based on 1.42 processor units corresponding to 1h of computing time. A computer hours matrix in terms of usage rate per instruction by phase is given in the section entitled Resource Allocation based on the above data.

The summary output from the cost estimation algorithm, SPREAD, includes: a) cost per routine based on either historic burden rates or proposed burden rates; b) average cost per instruction, number of instructions. and category; c) total number of development computer hours per routine; d) simple graphic display of schedule and events; e) cost breakdown by development phase per routine in total dollars and percent; f) cost breakdown by activity, by routine, and summed over all routines in the software, such as management, review, documentation, specifications, design, coding, and testing; and finally, g) manloading and cost summary by segment showing for each month the labor breakdown for senior staff, staff, technical, clerical; also, computer hours by month, other direct charges, cost by month, and cumulative cost.

The outputs from the cost estimation algorithm are considered a "trial set" for the cost estimation group. The trial set is used in combination with all other sources of data to test that cost position against the project objectives. The approved trial set, which contains the best judgmental and quantitative measures of cost per software element and per activity, becomes the input to the official pricing computer run. Necessary translation of the data set to exactly match the customer's work breakdown structure (WBS) or other appropriate cost elements is made for the final pricing run. The official cost figures are produced by cost guidelines, approved rates, and procedures that have been established by the in-house pricing group and approved by the government auditor. The results of the software cost estimation algorithm are retained as cost backup data and for possible use in later cost justification. The official cost figures from the pricing computer run show manloading and costs collected and aggregated against the customer's WBS (see Fig. 10).

This cost estimation approach has the following advantages. a) The amount of data required to obtain a cost estimate is minimal. b) The software designer immediately sees the cost consequences of his preliminary design. c) To the extent that the routine is directly traceable to a requirement, the requirement is directly traceable to a total cost. d) Comparisons of alternate schedules on resource allocation can be made rapidly. e) A cost justification is produced that can stand the test of government audit.

## SOFTWARE COST DATA BASE

Sensitivity coefficients or exchange ratios used by the cost estimation algorithm reside in the data base. The objective in cost estimating is to realize greater accuracy and precision while anticipating risk and its impact on profit. The greater the risk, for a given confidence level, the greater the allowance on the part of the estimator for this uncertainty. The more uncertain the estimate, the less competitive is the cost proposal. Further, an estimate of cost backed by relevant cost experience facts gives confidence to negotiator, management, and the customer. Cost justification is demanded by the customer, and is usually provided in the form of lower level back-up data.

Valid data based on proven experience are required for any cost estimation process. Producing good operational software is dependent on the many activities of good people properly directed and motivated. The process is difficult to measure and apply widely beyond the technology center for which the performance measures were derived. There is no universal model. Applying "average productivity rates" without knowledge of the development steps that were used in deriving the rates is wrong. Averages based on large samples are useful in comparison of a local variable against a global variable, if the estimator is reasonably certain he is comparing truly equiv-

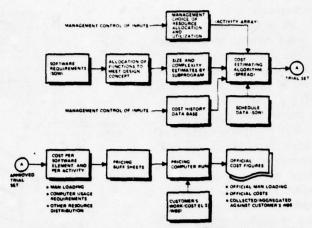


Fig. 10. Simplified costing sequence for TRW software, showing cost estimating algorithm data flow.

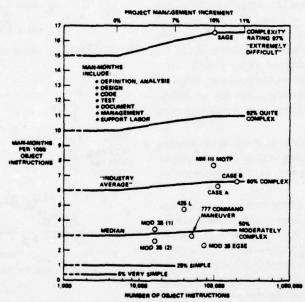


Fig. 11. Cost estimating graph, showing industry-wide averages.

alent measures. Metzelaar has studied the problem of industry-wide productivity rates in terms of program size and complexity, as shown in Fig. 11 [12].

The activities that productively occupy the analyst's time, along with the plant overhead and other direct charges, largely determine the price to the customer. It is in the company's best interest to agree on measurable activities that should be available in the software cost data base for cost justification and estimation of new work. Brief definitions of activities that have been identified in this regard are given below [2].

Learning and Orientation (L): Those actions necessary to gain understanding of the hardware and software to be applied in the project, and to learn the operating requirements, specifications, and constraints to be satisfied by the software package being produced.

2) Analysis and Design (A): All actions necessary to generate technical solutions, which are expected to satisfy requirements and specifications within the imposed constraints. It includes determining and evaluating technical approaches, determining and evaluating the effects of specific requirements and constraints on potential technical solutions, selecting the best solution, and describing that solution so it can be coded. This activity includes the rough documentation that is normally produced in generating a technical solution, but not the efforts of finalizing the documentation package.

3) Coding (C): The translation of the detailed steps of the technical solution into machine-readable language; the ordered list, in computer code or pseudocode, of the successive computer operations for solving a specific problem. It does not include actual input to the computer

(except for time-share inputs), compiling, nor testing the adequacy or validity of the instructions.

- 4) Test and Checkout (T): All actions necessary to assure that the instructions coded by the programmer cause the machine to do what the programmer intended it to do. It covers preparing the test data, compiling the program, running the test data, reviewing the compiler and/or machine output, identifying errors, correcting the errors, and making changes to the program that improve its reliability and efficiency. It does not include verifying that the software product satisfies the requirements and specifications stated by the customer or sponsor.
- 5) Verify/\u00faulity (V): Those actions necessary to assure that the software product satisfies the requirements and specifications provided by the customer or sponsor. It includes preparation of qualifying test data, verification runs, evaluation of computer run results, and adjustments to the program to correct any deficiencies.
- 6) System Integration and Test (I): All actions necessary to assure that the subsystem or module as programmed will interface with other subsystems or modules to provide a satisfactory system. It includes preparing data for evaluating the joint functioning of two or more subsystems or modules, making the evaluation runs, evaluating the joint operation, and making necessary modifications to the software in accordance with prevailing configuration control procedures.
- 7) Documentation (D): Those actions that require technical editing, technical typing, art work, and reproduction of deliverable documents to the customer or sponsor.

An example of the cost per instruction for the significant categories of software previously defined versus degree of difficulty is given in Fig. 12. The cost figures, in principle, include all seven development steps described in the section on software development and test management. Activities that are defined immediately preceding are inherently included in the cost figures as are all other direct charges up to and including general and administrative costs.

In late 1971, TRW compiled in-house data on a wide variety of software development characteristics, which in turn were independently analyzed by Lulejian and Associates under contract to the U.S. Air Force [13], [14]. The variables of particular interest for our present purposes were analyzed by Lulejian and Associates for a package of SS routines developed by TRW. A brief interpretation of four of the analyses that bear on the cost of developing software is presented next.

Consider Fig. 13. This is a plot of cost versus number of instructions for the 88 routines. Two conclusions can be drawn, either by direct observation or by regression analysis. a) There is a general trend in the data—larger routines cost more. b) A linear fit to the data is not very good. If one attempts to predict costs from routine size

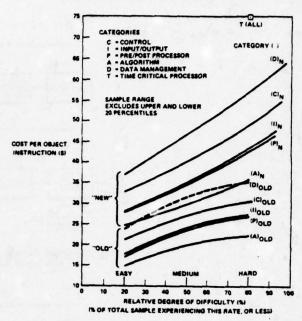


Fig. 12. Cost per object instruction versus relative degree of difficulty.

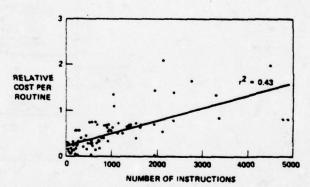


Fig. 13. Relative cost per routine versus number of instructions.

alone, one can expect errors of about 50 percent in the predicted cost. The prediction is better than nothing, but not much better. Fig. 14 shows the same data plotted another way—cost per 1000 instructions versus number of instructions. If the cost of the routine can be estimated by a fixed cost per 1000 instructions, one would expect this data to show constant cost. It appears to be constant, with a large error, especially for small routines. Again, one concludes that cost can be predicted from number of instructions provided that one is willing to accept fairly large errors.

Cost depends on a number of other factors, and one might hope to get better cost estimates by including difficulty, programmer experience, etc. A considerable number of fits were tried. The answers, unfortunately, were negative. Including other variables did not result in an equation that gives any significantly better fit to the data. Fig. 15 shows an example of one of the variables—

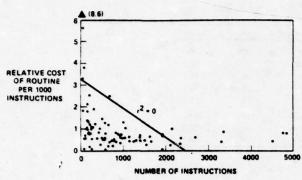


Fig. 14. Routine unit cost versus number of instructions.

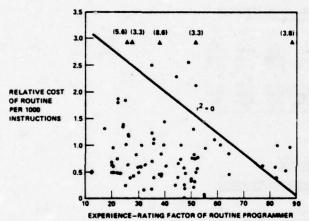


Fig. 15. Routine unit cost versus experience rating factor of programmer.

experience of programmer. A glance at the curve suggests that a knowledge of this variable does not help one predict cost. The regression analysis results support this suggestion. Table I shows the results of a multivariate linear regression analysis. Adding the remaining variables gives a better fit, but not much better.

The conclusion is that there are no simple universal rules for costing software accurately. It is necessary to understand the nature of the individual program and the individual routines within the program.

## RESOURCE ALLOCATION

Various researchers have separately discovered by empirical methods a rule of thumb, namely, that analysis and design account for 40 percent, coding and debugging account for 20 percent, and checkout and test account for 40 percent of the total resource (cost) spent for software development, the 40-20-40 rule. If the cost of "modeling" of physical systems is to be borne by the software development group, as contrasted to modeling analysis from raw data by technology centers within the company (such as thermal, attitude control and pointing, propulsion, electrical power, and so forth) an extra 30 percent typi-

TABLE I MULTIVARIATE REGRESSION SUMMARY

	EQUATIONS	OF FIT
COST = 0.25	+ 0.00028 (NUMBER OF INSTRUCTIONS)	,2 - 0.43
COST - 0.021	+ 0.00018 (NUMBER OF INSTRUCTIONS)	,2 - 0.49
	+ 0.19 (DIFFICULTY)	
COST = 0.005	+ 0.00029 (NUMBER OF INSTRUCTIONS)	r2 = 0.60
	- 0.00011 (NUMBER OF LOGICAL INSTRUCTIONS)	
	- 0.00027 (NUMBER OF INPUT/OUTPUT INSTRUCTIONS	
	+ 0.0086 (NUMBER OF INTERFACES)	
	+ 0.136 (DIFFICULTY)	
	+ 0.0034 (PERCENTILE RATING)	
*11	+ 0.0080 (YEARS OF EXPERIENCE)	
	· 0.0259 (YEARS OF SCF EXPERIENCE)	
	· 0.103 (SUPERVISORS RATING)	

TABLE II
Computer Program Development Breakdown

	ANALYSIS AND DESIGN	CODING AND AUDITING	CHECKOUT AND TEST
SAGE	39%	14%	47%
NTDS	30	20	50
GEMINI	36	17	47
SATURN V	32	24	44

cally would be required in addition to the software development by itself. In other words, the deriving of equations of motion or physical behavior of other physical systems from raw data is not included in the software development resource allocation. Similarly, the cost of computing time is not covered in the resource allocation. Experience has consistently shown that computer costs add an additional 20 to 25 percent of the total cost of the project; that is, a \$5 million/year software development contract will cost the government an additional \$1 million for GFE computing time to the developer. Boehm discusses his views on resource allocation in [15]; his findings are presented in Table II. The findings of an in-house survey by Metzelaar are presented in Table III. Independently derived resource allocation for the 1000 man-month costing example was given in Figs. 4 and 5.

The basic resource allocation data that are required for application of the cost estimation algorithm are shown in explicit form for purposes of illustration only in this paragraph. Table IV shows a simplified version of how "complexity" might be handled in a preliminary design. For example, if we are designing one routine in a large system that we estimate to have 1000 executable, or object, instructions (not Fortran IV source instructions), and that has as its primary function setting up the input

TABLE III
AVERAGE PERCENTAGE OF ANALYST'S TIME BY ACTIVITY

ACTIVITY	INFORMATICS	RAYTHEON	TRWª	AVG
ANALYSIS	20%	20%	20%	20%
DESIGN	16	20	20	18.7
CODE	16	25	24	21.7
TEST	32	25	28	28.3
DOCUMENT	16	10	8	11.3
	100%	100%	100%	100%
MIX: SCIENTIFIC	(60)	(0)	(100)	(53.3)
BUSINESS	(40)	(100)	(0)	(46.7

Substantial variation from project to project.

TABLE IV

Cost Per Instruction as a Function of Degree of Difficulty
(Example Only)

DEGREE OF			CATE	SORY		
DIFFICULTY	С	- 1	P	A	D	T
OE.	\$21.00	18.00	17.00	15.00	24.00	75.00
OM	27.00	24.00	23.00	20.00	31.00	75.00
ОН	30.00	27.00	26.00	22.00	35.00	75.00
NE	33.00	28.00	28.00	24.00	37.00	75.00
NM	40.00	35.00	34.00	30.00	46.00	75.00
NH	49.00	43.00	42.00	35.00	56.00	75.00

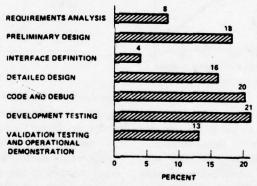


Fig. 16. Typical allocation of resources in custom software development and test.

to the main algorithm, we first categorize it as a "preprocessor," or category P. It is new and seems to be of medium difficulty (compared to others we have seen), therefore it is further categorized as new-medium, or NM. We enter on the worksheet for this routine its name, say PROG, its category P/NM, and a number of executable instructions equal to 1000. The consequence of this action is that PROG, at \$34.00/instruction, will cost the customer (i.c., directly timeable to PROG) \$34 000. This covers the cost of all activities, including analysis, design, documenting, coding, checkout, and test. These activities are spread into the development phases according to the typical allocation of resources shown in Fig. 16. Summing the first four phases prior to code and debug shows we allocate 46 percent of our total dollar there,

coding takes 20 percent, and the two major phases after coding take the remaining 34 percent. This distribution has been tested against at least two other computer program development cycles, provides a good fit to present design techniques, and will be adopted as the nominal.

The resource distribution in the seven phases [steps 1)-7) previously defined correspond to the seven phases A-G] will vary as a function of "category," and may be interpreted as a perturbation about the nominal. In all cases H, the O&M phase, will be treated differently since it is not characterized by the same objectives or activities as the development phases. The percent variation of total resource allocation as a function of category is given in Table V.

After classifying the software package (at the routine

TABLE V
Resource Allocation as a Function of Category for Each
Phase (Example Only)

				PHAS	E		
CATEGORY	A	8	c	D	E	F	G
c		20	2	20	14	24	12
1	8	18	4	15	21	21	13
P		19	3	14	23	21	12
A		18	1	21	23	17	12
D		22	4	14	16	24	12
T	8	11	12	10	24	21	14

level) by category and degree of difficulty, and defining the development phases for producing the software, we define the "activities" to be performed during each development phase. The software statement of work is the source authority for defining activities against which costs will be collected at an appropriate level in the work breakdown structure. The 25 activities (or tasks) that comprise the 7 development phases, which in turn lead to preparation of the deliverables for our hypothetical project, are shown in Table VI.

The assignment of resources to each of the 25 activities for each of the 7 development phases is shown in Table VII. Historical data from the cost data base combined with engineering judgment gained from work on previous similar projects are used in making the assignment.

The remaining step is to assign a properly time-phased computer resource to be used for the development and test of the software. The rationale for such a computer hours matrix was given earlier. The numerical results are given in Table VIII, together with an explanatory note on interpreting the data array.

An example in the application of the entire process would be useful. Within the space of this paper it is not practical; however, preliminary design of a large command and control software system was carried out in late 1971 that provides two useful insights for our present purposes. The first is that the command and control portion of the software system consisted of two major subsystems, the first containing 69 subprograms which were considered quasireal-time, and the second containing 29 subprograms which were considered real-time. In fact, some of the former contained real-time subprograms and the latter some nonreal-time subprograms. Thus, if a casual observer were to characterize the software as solely real-time (or nonreal-time), in hopes of deriving useful cost data base parameters, he would be certain to fail in precise interpretation of the performance data. The resource allocation in terms of burdened man-months and computer hours for the effort is given in Table IX at the module level, where routines make up modules, and modules make up the system. If this system had been carried out in a complete development cycle, a new data set would be entered into the cost data base and variances between actual and predicted resources would be available to aid in the next estimating effort.

A second insight that seems useful is the cost estimating relationships that emerge when the software elements are separated into meaningful groupings of, in this case, real-time, real-time plus its supporting environment (pre-and postprocessors programmed in higher order language, for example), and command and control elements that are not time-critical (see Fig. 17). The data are now highly correlated, and the differences in productivity rates between the groupings would be expected to range nearly 3 to 1. The observed differences should not be attributed to lack of correlation between dependent variables that are seen to be members of different sets.

## TOPICS FOR FURTHER WORK

The topics that were originally identified for discussion will be summarized, together with key issues that influence cost estimating of large software systems and that need further work and study.

1) What is the typical code production rate per programmer man-month? A working standard may be inferred from the data given of 1 object instruction/man-hour, which is equivalent to 156 instructions/man-month, or 1870 instructions/man-year, or 6.4 man-months/1000 object instructions for nontime-critical software.

If a burdened man-month varies between \$4300 and \$4900 in 1972 dollars for a typical programming department's labor mix, on the average this working standard translates to \$27.50-\$31.50 as the cost per instruction. Both case history A and B correlate strongly with these standards. They are characterized as large software jobs whose development is firmly structured and controlled. But what about the R&D work, not well structured and controlled, and which therefore does not lend itself to estimation by extrapolation from a historical cost data base?

The underlying question is whether we have identified all the right parameters to capture and quantify in our cost data base. Do we understand the causal relationships, and can they be isolated from their effects or symptoms? For example, is the SAGE data point where it is in Fig. 11 because it is "extremely complex" or because the requirements were poorly defined?

2) How does code production rate vary with problem complexity? With more sophisticated physical systems being brought under software command and control, a

TABLE VI ACTIVITIES AS A FUNCTION OF SOFTWARE DEVELOPMENT PHASE

DEVELOPM	ENT	PHASE A	PHASE 8	PHASE C	PHASE D	PHASE E	PHASE F	PHASE G	PHASE H
ACTIVITY	ASE	PERFORMANCE AND DESIGN REQUIREMENTS	IMPLEMENTATION CONCEPT AND TEST PLAN	INTERFACE AND DATA REQUIREMENTS SPECIFICATION	DETAILED DESIGN SPECIFICATION	CODING AND AUDITING	SYSTEM TESTING	CERTIFICATION AND ACCEPTANCE	OPERATIONS AND MAINTENANCE
MANAGEMENT	1	PROGRAM MANAGEMENT	PROGRAM MANAGEMENT	PROGRAM MANAGEMENT	PROGRAM MANAGEMENT	PROGRAM MANAGEMENT	PROGRAM MANAGEMENT	PROGRAM MANAGEMENT	PROGRAM MANAGEMENT
MANAGEMENT	2	PROGRAM CONTROL	PROGRAM CONTROL	PROGRAM CONTROL	PROGRAM CONTROL	PROGRAM	PROGRAM CONTROL	PROGRAM CONTROL	PROGRAM
REVIEWS	3		PRELIMINARY DESIGN REVIEW (POR)	INTERFACE DESIGN REVIEW (IDR)	CRITICAL DESIGN REVIEW (CDR)	SYSTEM TEST REVIEW (STR)		ACCEPTANCE TEST REVIEW (ATR)	OPERATIONAL CRITIQUES
DOCUMENTS		DOCUMENT AND EDIT	DOCUMENT AND EDIT	DOCUMENT AND EDIT	DOCUMENT AND EDIT	OOCUMENT AND EDIT		OOCUMENT AND EDIT	DOCUMENT AND EDIT
	5	REPRODUCTION	REPRODUCTION	REPRODUCTION	REPRODUCTION	REPRODUCTION		REPRODUCTION	REPRODUCTION
	6	REQUIREMENTS DEFINITION	POR MATERIAL	EVENT GENERATION INTERFACE	PRODUCT CONFIG DETAILED TECH DESCRIPTION	TECHNICAL DESCRIPTION UPDATE	PRODUCT CONFIG DETAILED TECH DESCRIPT UPDATE	REQUIREMENTS CERTIFICATION	SOFTWARE PROBLEM REPORTS (SPR)
REQUIREMENTS	,	REQUIREMENTS ALLOCATION	PERFORMANCE AND DESIGN	COMMAND DEFINITION I/F	(PART II) (WITHOUT LISTINGS)	TRAINING DOCUMENTATION	(PART II) (MITH LISTINGS)	POCUMENTATION UPDATE (PART II)	
SPECIFICATIONS	•		REQUIREMENTS -	TELEMETRY DEFINITION I/F			SPECIFICATIONS UPDATE		
	,			OPERATIONAL ENVIRONMENT I/F				- 6	
	10	TRADE STUDIES	TRADE STUDIES	TRADE STUDIES	TRADE STUDIES				
	11	INTERFACE REQUIREMENTS	FUNCTIONAL DEFINITION	DATA DEFINITIONS	ALGORITHM DESIGN	ALGORITHM UPDATE	PROGRAM UPDATE		
DESIGN 13	12	HUMAN INTERACTION	STORAGE AND TIMING ALLOCATION		PROGRAM DESIGN				
	STANGARGS AND CONVENTIONS	GATA BASE DEFINITION	DATA BASE DESIGN		DATA BASE UPDATE	DATA BASE UPDATE			
	14		SOFTWARE OVERVIEW (PRELIMINARY)		SOFTWARE OVERVIEW UPDATE				
CODING	15	= .			PROTOTYPE CODING	OPERATIONAL CODING			
	16		PRODUCT AND CONFIGURATION CONTROL	PRODUCT AND CONFIGURATION CONTROL	PRODUCT AND CONFIGURATION CONTROL	PRODUCT AND CONFIGURATION CONTROL	PRODUCT AND CONFIGURATION CONTROL	PRODUCT AND CONFIGURATION CONTROL	PRODUCT AND CONFIGURATION CONTROL
	17		DATA BASE CONTROL	DATA BASE CONTROL	OATA BASE CONTROL	DATA BASE CONTROL	DATA BASE CONTROL	DATA BASE CONTROL	DATA BASE CONTROL
	18	TEST REQUIREMENTS	TEST PLANS	INTERFACES	TEST PROCEDURES	DEVELOPMENT TESTING PLANNING	SYSTEM TEST PLANNING	ACCEPTANCE AND TEST PLANNING	INTEGRATION TESTING
CONFIGURATION CONTROL AND	19			15.00		DEVELOPMENT TEST	SOFTWARE SYSTEM TEST	ACCEPTANCE DEMONSTRATION	SPR CLOSURE
<b>GRA</b>	20						HARDWARE/ SOFTWARE SYSTEM TESTING		
	21	ACCEPTANCE TEST REQUIREMENTS	QUALITY AND RELIABILITY ASSURANCE PLANS	QA AND RA MONITORING	GA AND RA MONITORING	G AND RA MONITORING	Q AND RA MONITORING	Q AND RA MONITORING	Q AND RA
	22					TEST SUPPORT	TEST SUPPORT	TEST SUPPORT	TEST SUPPORT
	23								
	24		OPERATIONAL CONCEPT	OPERATIONAL TIMELINE	OPERATIONAL CONCEPT UPDATE	USER'S MANUAL (PRELIMINARY)	OPERATIONAL TIMELINE UPDATE	USER'S MANUAL UPDATE	INTEGRATION SUPPORT
OPERATIONS	25		TRAINING PLAN		TRAINING PLAN UPDATE	TRAINING	TRAINING	TRAINING AND REHEARSAL	TRAINING AND

corresponding increase in technological risk and complexity causes an increase in a responsible cost estimate. Reliable estimates place this increase in cost right now from 3 to 5.5 times the average of our historical cost data base. The leading causes of these predicted increases need serious attention and study, and effective management methods to detect and deal with them. For example, we

have already observed in two cases that the cost of producing real-time (or time-critical) software is three times more costly than nonreal-time software (see Figs. 12 and 17). Should the development dollar be spent on highly optimized machine-dependent code by the applications programmer or on more efficient compilers and assemblers?

Cost tradeoffs are needed between the crucial param-

TABLE VII

Cost Matrix Data Showing Allocation of Resources as a Function of Activity by Phase (Category P)

	- 5		-	PHASE				
ACTIVITY	A	8	2	0	1	£	9	H
	(8)	(19)	(3)	(14)	(23)	(21)	(12)	(0)
1	10	6		6	7	5	10	2
2	8	3	3	3	3	3	3	5
3		6	4	6		3	8	5
4	13		5	6	5	5	4	2
5	5	2	2	3	3	2	2	1
6	22	8	7	12	3	7	5	
7	10	8	7		2		•	
8	De Dei		7			5		
9			6					
10	17	10	10					
11	2	10	10	9	7	6		
12	2	5		13				
13	4	7	10		3	5		
14		4		3				
15				5	25			
16		4	4	5	4	4	10	10
17		3	6	5	6	5		10
18	5	6	3	8	5	2	5	10
19					10	15	14	5
20						10		
21	2	4	5	5	7	•	•	3
22					5	6		4
23								
24		4	3	2	3	5	3	25
25		2		1	2	3	5	10

TABLE VIII
COMPUTER HOURS MATRIX SHOWING COMPUTER USAGE ALLOCATION
AS A FUNCTION OF PHASE

PHASE			CATE	GORY		
FHASE	С	-1	P	A	D	T
A	0	0	0	0	0	0
B	0	0	0	0	0	0
C	0	0	0.0017	0.0007	0.0007	0.01
D	0.0017	0.0017	0.0017	0.0017	0.0017	0.01
E	0.007	0.005	0.008	0.007	0.007	0.04
F	0.008	0.005	0.01	0.007	0.007	0.04
G	0.005	0.004	0.005	0.004	0.004	0.005
H	0	0	0	0	0	0

eters that define the problem complexity and the cost benefits. A marginal utility theory is needed for the economics of software development. Parameters that are crucial in defining problem complexity include degree and time phasing of simulation, extent of prototype code, parallel development of different formulations of key algorithms, new computer hardware/software, assembly language coding, multisite operations and interchangeability, growth requirements, critical timing and throughput, hardware/software interfaces, fault-tolerant computing, core occupancy constraints, reliability, and safety [16]-[18].

3) How does this rate vary as a function of computer availability? Accessibility? Configuration? The President's Blue Ribbon Defense Panel reported in June 1970 in its staff report on automatic data processing, that because of the government's method of selection and procurement of

TABLE IX
RESOURCE ALLOCATION FOR LARGE COMMAND AND CONTROL
SOFTWARE SYSTEM (PRELIMINARY DESIGN)

MODULE	INSTRUCTIONS	MAN-MONTHS	COMPUTER UNITS x 1.43
CC (17)	21,760	162	425
CC(16)*	45,020	413	994
CC (14)	29,100	217	100
CC (7)	13,200	102	200
CC (15)	24,350	254	487
RT (4) **	3,000	44	- 4
RT (6) **	5,900	44	110
RT (16) **	16,920	256	805
RT (4) ***	2,560	61	200
ICS (9)	18,350	116	416
	190,780	1,002	4,556 ****

\* MODULE CONTAINS 4 REAL-TIME ROUTINES \*\* MODULES SUPPORT REAL-TIME FUNCTION \*\*\* MODULE ENTIRELY REAL-TIME \*\*\*\* SOUNALENT COMPUTER MOURS, 379/165

CC COMMAND AND CONTROL
RT REAL-TIME
ICS INTERPRETIVE COMPUTER
SIMULATOR
() NUMBER OF ROUTINES

LEGENO

Example: The column sum of category C, control routine, is  $217 \times 10^{-4} h$  instruction, or  $1.4 \min/\text{instruction}$ . The total resource required is then distributed over phases  $D\!-\!G$ , as shown above, e.g.,  $0.42 \min/\text{instruction}$  for phase E, coding and auditing.

computer systems, multiple computers in the same geographical area can result in costs that are as much as five times larger than would be necessary if a few large computers were used in a shared operating mode. The underlying problem here is that of data privacy and protection (security) for enabling the use of remote terminals by government agencies for both classified and unclassified work. An independent evaluation by E. C.

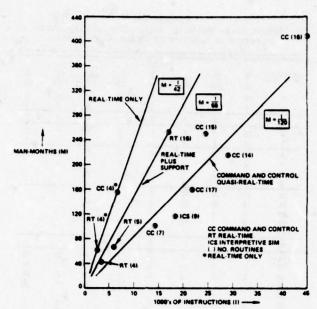


Fig. 17. Cost estimating relationships, showing wide variation within single command and control software system.

Nelson of TRW is that a dedicated computer installation costs three to five times as much to operate as if it were a time-shared installation with proper remote terminals. If private industry can enjoy these benefits of cost effectiveness by use of remote terminals (three to five times less expense), it would seem that some, if not a large part, of government installations could enjoy the same benefits if the problems of data privacy and protection were solved to everybody's satisfaction.

A management goal is to do the difficult parts of the software twice—once to produce prototype code early in the development cycle using perhaps 8–10 percent of the resources and an improved version with the technological risk eliminated for the operational version. One pitfall is that the risk is highest in procurements where a new computer and its operating system are being developed concurrently with advanced applications software, so just when computer time is needed the most, it is available the least. More planning to reduce the computer selection and procurement time is needed.

Furthermore, in some operational systems, the computer configuration available to the user is significantly different from the computer configuration available to the computer program associate contractor. Such a working environment requires an integrating contractor to maintain system-level configuration control. This may add 10 percent to the total cost and is viewed as an insurance policy. To eliminate such occurrences, standards for computer configurations are needed, and configuration control is needed by the associate contractor and, equally importantly, by the user.

4) What are the pieces of information required to make a

realistic prediction of software development cost? We have identified the activities that occupy the analyst's time in a 25 × 7 matrix (Table VI) and elsewhere. However it is one thing to identify significant activities for allocation of resources, and it is another thing to hold the right set of individuals accountable for the expediture of those resources for those things over the life cycle of the software development, which spans 12-24 months or longer. The purpose of a cost estimating system is to reduce the variance between estimates of what a software development should cost and what it actually does cost. The cost history data file is a key element in an improved cost estimating system. Further work is needed in capturing and maintaining historical estimates, costs, and quantitative facts to provide more effective data for estimating future costs and sizes. Further work is needed in maintaining cost control over the actual performance period considering the following.

a) Recording and displaying of both individual and aggregate programmer activities, such as previously defined by L,A,C,T,V,I, and D, by easy-to-use automated project control and prediction tools.

b) Fast computation of project cost- and time-tocomplete predictions by current and look-ahead activity statistics, including uncompensated overtime, as is now feasible through automated cost control tools.

c) Provision for storing, retrieving, and tracing of project statement of work deliverables to their cost estimate and cost estimate to actual cost, by activity, over the useful life of the software through cost control tools.

5) How does code production rate vary as a function of

programmer quality? Quality requirements on the final product? The point to be made here is that we have seen convincing evidence of the validity of the 40-20-40 rule. For a \$5 million project, \$2 million will be spent on checkout and test to the extent the rule applies. A management objective is to check out every logic path in the software to deliver an error-free program. This is a high-quality final product insofar as being error-free is a measure of quality. Achieving this objective may add significantly to the development cost but reduce even more significantly the operations and maintenance cost. With improved test tools and flow measurement tools, considerable improvement in efficiency of more branches checked per computer hour becomes realizable. From the analysis given for one software system, the quality of the programmer in years experience and supervisor's rating seemed to make no difference in the cost in terms of the goodness of fit, the adjusted index of determination  $(r^2)$  given in Table I and Fig. 15. As R. J. Hatter says, some of the signs exhibit counter-intuitive behavior.

Technical performance standards and criteria are needed in the software development process. A wrong standard is better than no standard because at least it gives a sense of direction to the effort. A technical performance measurement system is needed for applying to the software development process, and which is understood and applied at all management and performer levels over the life cycle. Since software development is more process oriented than product oriented, such standards become measures of group productivity if not individual productivity. In some organizations it may be interpreted to be against company policy to measure and report on a productivity index of an individual member of the professional staff, such as code production rate, or error rates (as measured, for instance, by software problem reports). Producing quality custom software is a profit-motivated business, and more businesslike procedures are being applied as the product lines mature [19].

6) How does cost vary with completeness of problem formulation? Using the 40-20-40 rule again, but this time dealing with the "analysis and design" 40 percent (which for our example of Fig. 16 was 46 percent), we have evidence that analysis and design takes the lion's share of the software development dollar. A management goal that has proven effective in the past is to produce superior documentation which is reviewed with a knowledgeable technical staff in the customer's complex. Thorough and continuous involvement of the customer in the development process has been a reality of several large software developments. Nothing takes the place of competence and communication when it comes to understanding the customer's or sponsor's requirements. Translating total system requirements, which may not be well understood even by the customer, to the software system requirements is a crucial first step. In the requirements definition and

preliminary design phase, there is typically a lack of data but lots of leverage (influence on the ultimate design outcome), in contrast to the operational phase where there are lots of data but little, if any, leverage. This is a significant area for cost- and performance-effective improvement.

7) What is the role of "design-to-cost" in the development of large-scale software? The DOD joint logistics commanders have recently entered into an agreement concerning the acqusition and ownership of major weapon systems based on the concept of "design-to-cost" [20]. The term "design-to-cost" is defined as a process utilizing unit cost goals as thresholds for managers and as design parameters for engineers. Cost parameters (goals) are to be established that translate into "design to" requirements. In the past, the order of priorities for major weapon system acquisition and ownership has been: a) performance, b) availability, and c) cost. Today, under the design-to-cost doctrine, the order is: a) cost, b) performance, and c) availability. The applicability of the design-to-cost concept to hardware is now beginning to emerge and is being reduced to practice. The relationship of the design-to-cost concept to software is not now understood.

In the 1975-1980 era, this reordering of priorities for major weapon systems, of which software is generally an integral part, is expected to have a sizeable influence on both hardware and software life cycle acquisition and ownership, particularly in the sense that the unit cost requirement will become a driving parameter in the design of large-scale software. System development will be continuously evaluated against the unit cost requirements with the same rigor as now applied to technical requirements. Practical tradeoffs must be made between system capability, cost, and schedule. Traceability of estimates and costing factors, including those for economic escalation, will have to be maintained.

#### ACKNOWLEDGMENT

The author wishes to thank Dr. R. H. Pennington, Chief Scientist of System Development Corporation, who invited the author to prepare this basic paper and to serve on a panel at the IEEE International Convention in New York, N. Y., March 20-23, 1972. The other panel members were T. E. Climis of IBM, V. LaBolle of the Los Angeles Department of Water and Power, and Dr. R. E. Merwin of the Safeguard Systems Office, each one having a different approach to the problem of the cost of developing large-scale software [21]. The author also wishes to thank many close TRW colleagues for their help in getting the ideas and material together for this paper, particularly, D. J. Alley, C. A. Bosch, W. V. Buck, R. D. Kennedy, W. L. Hetrick, W. A. Krumreich, W. C. Lynam, P. N. Metzelaar, Dr. D. D. Morrison,

Dr. F. J. Mullin. Dr. E. C. Nelson, F. Putich, E. A. Rollin, and L. L. Wolfson. A special note of recognition is due Dr. W. W. Royce, who was the prime originator of TRW's SPREAD cost estimation algorithm. The author must, in the final analysis, take full responsibility for this paper, including errors of fact, omission, or expressions of opinion.

The author wishes to thank R. J. Hatter of Lulejian Associates, Inc., for the release of Figs. 13-15 and Table I from the West Coast Study Group report, and Dr. B. W. Boehm of TRW, formerly of The Rand Corporation. for his resource allocation data, Table II [15], [22].

Guidance from the three referees and Dr. R. A. Short of the IEEE TRANSACTIONS ON COMPUTERS is appreciated. Their decision to depart from the traditional IEEE TRANSACTIONS ON COMPUTERS editorial policy and publish a software technical management paper is gratefully acknowledged by the author. Their aim is to provide information useful to the hardware computer systems designer in learning more about the software development process, in pointing to the tasks where the programmer expends his major effort and, it is hoped, in identifying problem areas to attack so future computer systems may ease the programmer's task.

#### REFERENCES

J. M. Beveridge, The Anatomy of a Win, J. M. Beveridge and Assoc., Inc., Playa Del Rey, Calif., 1970.
 W. C. Lynam, P. N. Metzelaar, and D. H. Hibsman, "S&ISD cost estimating system final report," TRW Systems Group, Redondo Beach, Calif., Nov. 13, 1970.

S. Stimler, Real-Time Data Processing Systems: A Methodology for Design and Cost/Performance Analysis. New York: McGraw-Hill, 1969.

[4] W. F. Sharpe, The Economics of Computers. New York: Columbia Univ. Press, 1969.
[5] G. F. Weinwurm et al., On the Management of Computer Programming. Princeton, N. J.: Auerbach, 1970.
[6] W. W. Royce and E. A. Rollin, "A software cost estimation Technique." TRW Systems Group, Redondo Beach, Calif., Oct. 27, 1979. 1970.

Oct. 27, 1970.

J. L. Martin, Jr., "A specialized incentive contract structure for satellite projects," Space and Missile Systems Office, Los Angeles, Calif., Apr. 18, 1969.

H. E. Boren and H. G. Campbell, "Learning curve tables," Rand Corp., Santa Monica, Calif., Rep. RM-6191-PR, Apr. 1970.

[9] R. H. Brandon, Management Standards for Data Processing. New York: Van Nostrand, 1963.
[10] C. P. Lecht, The Management of Computer Programming Projects, Amer. Management Assoc., New York, 1967.
[11] F. J. Mullin, "Computing time usage," TRW Systems Group," Redondo Beach, Calif., May 18, 1971.
[12] P. M. Metaleas, "Crest estimation, graph", TRW Systems

[12] P. N. Metzelaar, "Cost estimation graph," TRW Systems Group, Redondo Beach, Calif., April 30, 1971.
[13] C. A. Bosch and B. W. Boehm, "Software development characteristics CCIP-85 study group," TRW Systems Group, Redondo Beach, Calif. Oct. 3 (1971).

Beach, Calif., Oct. 8, 1971.
[14] R. J. Hatter, "CCIP study regarding analysis of TRW software analysis data (Excerpt)," Lulejian and Assoc., Inc., Redondo Beach, Calif., Nov. 29, 1971.

[15] B. W. Boehm, "Some information processing implications of air force space missions: 1970-1980," Rand Corp., Santa Monica,

Calif., Jan. 1970.
[16] H. Hecht. "Figure of merit for fault-tolerant space computers,"

I. Heent, "Figure of merit for failt-tolerant space computers,"
 *IEEE Trans. Comput. (Special Issue on Fault-Tolerant Computing)*, vol. C-22, pp. 246-251, Mar. 1973.
 G. J. Schick and R. W. Wolverton, "Assessment of software reliability," presented at the 11th Annu. Meeting German

reliability," presented at the 11th Annu. Meeting German Operations Res. Soc.. Hamburg, Germany, Sept. 6-8, 1972.
[18] J. R. Brown and H. N. Buchanan, "The quantitative measurement of software safety and reliability." TRW Systems Group, Redondo Beach, Calif., Site Defense Program Rep. SDP 1776, Aug. 24, 1973. ug. 24, 1973.

[19] E. R. Mangold, Software Development and Configuration Management Manual, Software Product Assurance, TRW Systems Group, Redondo Beach, Calif., Oct. 1973. H. A. Miley, Jr., I. C. Kidd, Jr., J. J. Catton, and S. C. Phillips,

"Joint design-to-cost guide, a conceptual approach for major weapon system acquisition," presented at the Soc. American Value Engineers Conf. Practice of Design-to-Cost, Long Beach,

V. La Bolle, "Cost estimating for computer programing," in Proc. IEEE 1972 Int. Conv. Exposition (New York, N. Y., Mar. 20-23, 1972). Contains large bibliography of 99 annotated

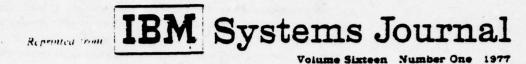
references, pp. 28-30.
[22] B. W. Boehm, D. W. Kosy, and N. R. Nielsen, "ECSS: A prospective design tool for integrated information systems," presented at the American Institute of Aeronautics and Astronautics. Integrated Information Systems Conf., Palo Alto, Calif., Feb. 17-19, 1971, Paper 71-228.



Ray W. Wolverton (S'48-A'50-M'52) was born in Hopkins, Mo., on October 20, 1924. He received the B.S.E.E. degree from Stanford University, Stanford, Calif., in 1950, and did graduate work in mathematics, controls, and computer system design at the University of Southern California, Los Angeles, and the University of California, Los Angeles, in 1950-1952 and 1952-1955, respectively; and in management theory at the Industrial College of the Armed Forces, Fort Lesley,

U. McNair, Washington, D. C. in 1971-1972. From 1950 to 1955 he was a member of the technical staff at Hughes Aircraft Company, working in the communication, navigation, and landing activity for the MX-11 79 manned interceptor. He joined the Ramo-Wooldridge Guided Missile Research Division in 1955, where his initial interests were in closed-loop trajectory simulations and orbital operations. Since 1955 he has been with the TRW Systems Group, Redondo Beach, Calif., working in the fields of flight mechanics, real-time command and control operations, computer software development and test management, and project management. His present interests are in large-scale computer program management, cost modeling and forecasting, and reliability (error-free software). Currently, he is a Senior Staff Engineer with the Systems Design and Integration Operations, Systems Engineering and Integration Division, TRW Systems Group. He was a coauthor and editor of the Flight Performance Handbook for Orbital Operations, produced under contract to NASA (New York: Wiley, 1963).

He is a member of Sigma Xi, the Scientific Research Society of North America, Phi Kappa Phi, the American National Standards Institute, and various professional societies. He is listed in Who's Who in the West and is a recipient of both NASA's Apollo Achievement Award and the astronaut's Silver Snoopy Award.



with permission from IBM Corporation

A method of programming measurement and estimation by C. E. Walston and C. P. Felix

Improvements in programming technology have paralleled improvements in computing system architecture and materials. Along with increasing knowledge of the system and program development processes, there has been some notable research into programming project measurement, estimation, and planning. Discussed is a method of programming project productivity estimation. Also presented are preliminary results of research into methods of measuring and estimating programming project duration, staff size, and computer cost.

# A method of programming measurement and estimation

by C. E. Walston and C. P. Felix

New materials technologies and architectures are significantly affecting computing system hardware. While the cost per bit of storage and the execution cost per instruction have both been decreasing, the same trend has not been true for software. Since software development has continued to be a people-oriented activity, a higher percentage of the cost to acquire a computing system is accruing to software development.

New management and programming techniques have been developed to improve programming efficiency. Among the improved programming technologies are the following:

- Chief programmer team, a programming organization built around a chief programmer, a backup chief programmer, and a librarian that effectively produces code in a disciplined and open environment.<sup>2</sup>
- Hierarchy plus Input-Process-Output (HIPO), a graphic design and documentation method that is used to describe program functions from the topmost level to great detail.<sup>3</sup>
- Development support library, a tool that provides current information about project programs, data, and status.
- Structured programming, a programming method based on the mathematical Structure Theorem<sup>3</sup> that enables programmers to understand and enhance programs that have been written by others,<sup>9</sup> as well as one's own programs.
- Design and code inspections, a review of program design, code, and documentation to detect errors prior to program execution.
- Top-down development, an ordering of program development and testing that begins at the topmost functional level and proceeds decrementally to the lowest functional level.

54 WALSTON AND FELIX

IBM SYST J

This paper discusses research into programming measurements, with emphasis on one phase of that research: a search for a method of estimating programming productivity. The method we present is aimed at measuring the rate of production of lines of code by projects, as influenced by a number of project conditions and requirements. We do not, however, measure the performance of individual programming project members. As we continue our research, we are continuing to learn more about the attributes of the programming process, about programming itself, and about better ways of analyzing the data.

Before starting the programming measurements research reported here we analyzed the literature on programming measurements and productivity that includes the work of Aron. Weinwurm and Zagorski, and Nelson. We also wanted to isolate the effects of improved programming technologies from the effects of monetary inflation, variations in computing cost, and ambiguities in the definition of computing quantities (such as that of the size of the delivered program product).

The software measurements project began in 1972 when we decided to assess the effects of structured programming on the software development process. To do that, a rigorous program was established to measure the then current software methodologies so that changes that result from the introduction of new methodologies could be measured. The initial phase of the measurements program was to identify variables to be measured, to design a questionnaire (called the Programming Project Summary questionnaire), and to develop a system for processing the data to be collected. The first report entered the system on January 23, 1973, and data have continued to be received and entered into the system from that time on. Since the establishment of the Programming Project Summary, two new reporting formats have been designed: the Software Development Report and the Software Service Report.

We understood at the outset that the established objectives might change as the project matured and as data were accumulated. Current objectives of the project discussed in this paper are the following:

objectives

- Provide data for the evaluation of improved programming technologies.
- · Provide support for proposals and contract performance.
- Gather and preserve historical records of the software development work performed.
- Provide programming data to management.
- Foster a common programming terminology.

NO. 1 · 1977

PROGRAMMING MEASUREMENT AND ESTIMATION

55

Table 1 Software project measurements reports

Report Name	Nature of the Report			
Programming Project Summary	Detailed report on the software development environment, the product (including errors), resources, and schedule.			
Software Development Report	Detailed report on the software development environment, the product (including changes errors), resources and schedule.			
Monthly Software Development Report	One report on product, resource, and schedule status. Changes in software development environment are noted.			
Software Service Report	Report on project size and on the software service environment.			
Quarterly Software Detailed data on product being serviced. Service Report resource status, and changes in software environment.				

Key to a software measurement program are the analysis of measurements and feedback of results to the suppliers of the raw data and to management. This paper discusses the data reporting and analysis in the software measurements program of the IBM Federal Systems Division. Described first are the measurements data base, the services that are available to users of the data, and descriptive statistics from the data base. The next section covers the analysis of programming productivity and describes a productivity estimation technique. Following this, the results of other analyses that have been performed with respect to factors such as documentation, staffing, and computer costs are presented. The last section briefly describes analysis efforts that are being contemplated for future research. In the measurements project discussed here, a sufficiently large quantity of data is being collected so that a programming project measurement system is used to provide the necessary flexibility and capability of storage control and analysis. The Programming Project Measurement System is briefly described in the Appendix.

## Messurements data base

Data contained in questionnaires submitted by line projects at prescribed reporting periods are stored in a computer data base where they are accessible for answering queries, preparing reports, or for analytical studies.

The basic measurements data base is structured so that it contains all the reports received, as described in Table 1. Each Pro-

gramming Project Summary milestone report is given a unique number when received and is assigned a separate record in the file. Project milestones are the following: start of work: preliminary design review: midway through software development: acceptance test completion: and every three months during the maintenance or service phase. Each initial and final Software Development and Software Service report is also filed as a separate record.

The monthly Software Development Reports plus any changes to the initial submission constitute separate records, as do the quarterly Software Service Reports and any changes to the initial Software Service Report. Each record is structured into fields, or variables, that correspond to the question response fields in the questionnaires.

Sixty completed software development projects are now in the data base. These completed projects, which represent a wide variety of programming technology, are summarized in Table 2. Their delivered source lines of code range from 4000 to 467000, and their effort ranges from 12 to 11758 man months. These programs include real-time process control, interactive, report generators, data-base control, and message switching programs. Some of the programs have severe timing or storage constraints and other programs have both types of constraints. Twenty-eight different high-level languages and 66 different computers are represented in the listing in Table 2.

With the previously described data base and with the capabilities provided by the Programming Project Measurement System, a wide variety of services can be provided to line project personnel. Queries can be answered, analyses performed, and programming project productivity estimation provided for new or on-going projects. Examples of services are discussed throughout the remainder of this paper.

Queries that can be answered by direct retrieval of data from the data base are of two general types. First is a request for data about a specific project. In this case, the Programming Project Measurement System generates a report that lists the answers to the questions submitted by personnel of the specified project. The second type of query is a more general request for information, an example of which is, "What has been the utilization of improved programming technologies by projects in the data base and what was the effect on project productivity?" The answer is provided in the form of a listing of productivity data sorted by project. The following breakdown indicates the types of reported and derived data that can be provided in response to a general inquiry:

NO. 1 · 1977

PROGRAMMING MEASUREMENT AND ESTIMATION

57

Table 2 Characterization of programs in the programming project data base

#### A. Small less-complex systems Batch storage and retrieval Batch inventory Batch information management Batch languages preprocessor and information management **Batch** reporting Batch financial information Batch scientific processing simulation **Batch** utility Batch operating system exerciser B. Medium less-complex systems Special-purpose data management (2) Batch storage and retrieval (2) Process control simulation Batch reporting Batch data-base utility (2) On-line scientific processing simulation Batch on-line scientific information management On-line business information management On-line storage and retrieval Batch hardware test support Batch scientific algorithm feasibility (3) Interactive scientific processing (2) System test support Batch planning (3) Batch military information management Special-purpose operating system C. Medium complex systems Real-time, special-purpose system exerciser Special-purpose operating system (2) Batch information modification **Batch** information conversion Data management Sensor-based mission control On-line scheduling Sensor-based mission simulation Interactive scientific processing Process control (3) On-line graphics System performance monitoring and measurement (3) Terminal data management Interactive information conversion Operating system extensions D. Large complex systems • Sensor-based mission monitoring and control · Interactive information acquisition Process control Sensor-based system exerciser (3) Sensor-based mission processing and communication (2)

## Programming project data:

Number of lines of delivered source code ordered by project.
 (Source lines are 80-character source records provided as input to a language processor. Job control languages, data definitions, link edit language, and comment lines are included. Reused code is not included.)

Table 3 Programming project descriptive data report for completed projects

	Median 50%	Quartiles 25-75%
Productivity Source lines per man month of total effort	274	150-440
Product		
Source lines (thousands)	20	10-59
Percentage of code lines not to be	5	0-11
delivered	69	27-167
Documentation per thousand lines of source code (pages)		
Resource		
Computer cost (as a percentage of project cost)	18	10-34
Total effort (man months)	67	37-186
Average manning	6	3.8-14.5
Effort distribution of preliminary design review (percent)	18	11-25
Distribution of effort (percent)		
Management and administration .	18	12-20
Analysis	18	6-27
Programming and design	60	50-70
Other	4	0-6
Duration (months)	11	8-19
Development error detection		
Errors per thousand source lines	3.1	0.8-8.0
Incorrect function	76	50-86
Omitted function (percent)	8	0.22
Misinterpreted function (percent)	17	7-25
Errors per programming man month	0.9	0.3-2.4

- Pages of documentation (including program source listings) delivered.
- · Source languages used to develop code.

## Resource data:

- Total effort (in man-months, including management, administration, analysis, operational support, documentation, design, coding, and testing) required to produce the lines of source code by project.
- · Duration of project in months.

Use of improved programming technologies (expressed as a percentage of code developed using each technique):

- Structured programming.
- · Top-down development.
- Chief programmer team.
- · Design and code inspections.

Table 4 Data for completed service projects

	Median 50%	Quartiles 25-75%
Product		
Lines of maintained source code (thousands)	103	56-474
Ratio of developed to maintained code	0.04	0-0.19
Resources		
Average manning per project	6	4-11
Maintained source lines per man (thousands)	15	5-24
Distribution of effort		
Management and administration (percent)	15	10-20
Analysis (percent)	10	4-30
Programming (percent)	72	40-80
Other (percent)	3	0-7
Duration (months)	18	11-31
Errors detected		
Errors per thousand lines of maintained code	1.4	0.2-2.9
Incorrect function (percent)	73	
Omitted function (percent)	- 11	
Misinterpreted function (percent)	13	

Table 5 Descriptive data from on-going projects

Programming	Median 50%	Quartiles 25 - 75% 5.4 - 30	
Source lines (thousands)	12.5		
Percentage of code lines not to be delivered	2.6	0-11	
Effort (man months)	72	30-205	
Distribution of effort:			
System analysis (percent)	15	10-20	
System design (percent)	20	15-25	
Code and unit test (percent)	30	20-35	
Integration and test (percent)	20	15-30	
Other (percent)	5	0-10	
Duration (months)	12	9-18	
Service	Median 50%	Quartile: 25 - 75%	
Maintained source lines (thousands)	148	55-340	
Ratio of developed to maintained code	0.05	0-0.09	
Effort (man months)	88	27-185	
Average manning (persons)	5	3-19	
Duration (months)	10.5	6.5-12	

#### Derived data:

- Productivity achieved, ordered by project. Programming
  productivity is defined as the ratio of the delivered source
  lines of code to the total effort (in man-months) required to
  produce the lines of code, and is computed from product and
  resource data.
- Average number of people required to work on the project.
   computed by dividing the total effort in man-months by the duration of the project in calendar months.

More complex and extensive search and analysis questions can also be answered. These are supported by a question analysis subsystem of the Programming Project Measurement System, which incorporates a statistical package for the manipulation and statistical analysis of many types of data. The more complex requests can be grouped into two types, descriptive and analytical. A descriptive request requires searching the data base for specific variables or derived variables and computing characteristics about their distributions such as the mean, mode, and standard deviation, in order to prepare the report. Table 3 illustrates one such report for the completed projects in the data base.

Because of the variability in the measurement data, the statistics in Table 3 are presented in terms of medians and quartiles. The median for the size of the delivered software product is 20 thousand lines and fifty percent of the projects reported that the sizes of their delivered source code ranged from 10 thousand to 59 thousand lines. The effort for software development reported was distributed into the major categories as shown. The error detection section of the table shows the distribution of errors reported during the development phase.

Table 4 provides descriptive statistical data for completed service projects. Most service activity is not purely maintenance, but includes development efforts as well. The ratio of developed source lines of code to maintained lines of code was 4 percent at the median.

Table 5 presents data on on-going programming and service projects. Since only a small percentage of these projects have been completed at this time, the statistics represent a mixture of actual measurements and estimates at various stages of completion.

## **Productivity analysis**

We have identified five major parameters that can help programming project personnel make estimates. These parameters, pro-

NO. 1 · 1977

PROGRAMMING MEASUREMENT AND ESTIMATION

ductivity, schedule, cost, quality, and size, are listed in order of increasing difficulty and complexity of analysis. Some of the difficulties arise from a lack of detailed data in the data base, as in the case of schedule data. Complexity of the quantitative data can create other difficulties. One significant difficulty is in identifying and measuring independent variables that can be used to estimate the desired variable, as is the case in estimating the size of the product to be delivered.

Productivity, which can be defined in terms of the quantitative measures that are in the data base, is a vital factor in all software estimating processes and, therefore, is of immediate value to project personnel. For this reason, the analysis performed to date has focused on productivity estimation. Productivity has often been defined as the ratio of output to input. Programming productivity is defined here as the ratio of the delivered source lines of code (DSL) to the total effort in man-months (MM) required to produce that delivered product.

The basic relationship between delivered lines of code and effort is shown in Figure 1. Each plotted point represents the data reported by a completed project in the data base. A number on the chart indicates a position where a number of data points are grouped sufficiently close together that they cannot be individually identified on the plot. The data have been plotted in the log-log domain so that they become approximately linear. The linear coefficients become power relationships when transformed back to the original domain of the data. The least squares fit to the data as plotted in Figure 1, yields the result:

 $E = 5.2L^{0.91}$ 

where

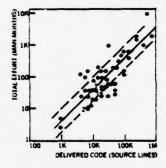
E = total effort in man months

and

L = thousands of lines of delivered source code.

productivity index

This relationship is nearly a first-power (or linear) relationship between effort and product size. The dashed lines indicate the standard error of the estimate on either side of the least squares fit. To identify the sources of scatter or variation of Figure 1, those variables that are related to productivity have been investigated. Preliminary findings have led to the development of a productivity estimator that provides an on-line capability to support proposed as well as on-going projects. A set of sixty-eight variables was selected from the data base, and those variables were analyzed to determine which were significantly related to productivity. Twenty-nine of the variables showed a significantly high correlation with productivity and have there-



2 WALSTON AND FELIX

IBM SYST J

fore been retained for use in estimating. Table 6 lists these variables and the reponses associated with them. To illustrate the meaning of Table 6, consider the first entry, which is derived from a multiple-choice question that asks the information supplier to circle his response to the following statement: Customer interface is (less than, equal to, greater than) normal complexity.

When the mean productivity was computed for all the completed reports in the data base that indicated less than normal customer interface complexity, the result obtained was 500 delivered source lines of code per man-month of effort (DSL/MM). By a similar computation, the mean productivity for all projects that reported normal complexity was 295 DSL/MM, and the mean productivity for those reporting greater than normal complexity experience was 124 DisL/MM. The change in productivity between less-than-normal and greater-than-normal customer interface is 376 DSL/MM, as noted in the final column in Table 6. Three variables in the table (overall personnel experience, code complexity, and design constraints) were formed by combining the answers to several questions in the questionnaire. It should be noted that this analysis was performed on each variable independently and does not take into account either the possibility that these variables may be correlated, or that there may be interrelated effects associated with them.

The twenty-nine variables were then combined into an index, based on the effect of each variable on productivity, as indicated by the above analysis, to form a productivity index. The productivity index is computed as follows:

$$I = \sum_{i=1}^{29} W_i X_i,$$

where

I = productivity index for a project

 $W_i$  = question weight, calculated as one-half  $\log_{10}$  of the ratio of total productivity change indicated for a given question i

 $X_i$  = question response (+1, 0 or -1), depending on whether the response indicates increased, nominal, or decreased productivity

An index was computed for fifty-one projects, and a plot of actual productivity for each project versus the computed productivity index and the least squares fit to this relationship is shown in Figure 2. The standard error of the estimate (standard deviation of the residuals) is shown as dashed lines.

To support project estimates, a shortened version of the data collection form is used that contains excerpted questions associated with the twenty-nine variables used in the index. A

rapid estimates

NO. 1 · 1977

PROGRAMMING MEASUREMENT AND ESTIMATION

63

Table 6 Variables that correlate significantly with programming productivity

Question or Variable	e odeni	Response Group Mean Productivit (DSL/MM)		Productivity Change (DSL/MM)
Customer interface complexity	Normal 500	Normal 295	> Normal 124	376
User participation in the definition of requirements	None 491	Some 267	Much 205	286
Customer originated program design changes	Few 297		Many 196	101
Customer experience with the application area of the project	None 318	Some 340	Much 206	112
Overall personnel experience and qualifications	Low 132	Average 257	High 410	278
Percentage of pro- grammers doing devel- opment who participated in design of functional specifications	< 25% 153	25 – 50% 242	> 50% 391	238
Previous experience with operational computer	Minima 146	Average 270	Extensive 312	166
Previous experience with programming languages	Minima 122	Average 225	Extensive 385	263
Previous experience with application of similar or greater size and com- plexity	Minima 146	Average 221	Extensive 410	264
Ratio of average staff size to duration (people/month)	< 0.5 305	0.5-0.9 310	> 0.9 173	132
Hardware under concurrent development	No 297		Yes 177	120
Development computer access, open under special request	0% 226	1-25% 274	> 25% 357	131
Development computer access, closed	0-109 303	6 11-85% 251	> 85% 170	133
Classified security envi- ronment for computer and 25% of programs and data	No 289		Yes 156	133

program, running in the Time Sharing Option of OS/VS (TSO) was developed to compute and list the index estimates. This terminal-based program allows rapid response to project requests for information. The estimate of expected productivity is returned to

Question or Variable	Response Group Mean Productivity (DSL/MM)			Productivity Change (DSL/MM)
Structured programming	0-33%	34-66%	66% 301	132
Design and code inspec-	0-33%	34-66% 300	> 66%	119
Top down development	0-33% 196	34-66 % 237	> 66% 321	125
Chief programmer team usage	0-33% 219	34-66%	> 66% 408	189
Overall complexity of code developed	< Average 314		> Average 185	129
Complexity of application of processing	< Average 349	Average 345	> Average 168	181
Complexity of program flow	< Average 289	Average 299	> Average 209	80
Overall constraints on program design	Minimal 293	Average 286	Severe 166	107
Program design constraints on main storage	Minimal 391	Average 277	Severe 193	198
Program design constraints on timing	Minimal 303	Average 317	Severe 171	132
Code for real-time or inter- active operation, or exe- cuting under severe timing constraint	< 10% 279	10-40% 337	> 40% 203	76
Percentage of code for delivery	0-90% 159	91 - 99% 327	100% 265	106
Code classified as non- mathematical application and I/O formatting programs	0-33% i 188	34-66% 311	67-100% 267	79
Number of classes of items in the data base per 1000 lines of code	0-15 334	16-80 243	> 80 193	141
Number of pages of de- livered documentation per 1 000 lines of delivered code	0-32 320	33-88 252	> 88 195	125

the requester in the form of a report that contains a comparison between the project estimate and the one derived from the data base. Also included is a list of the reported attributes or variables that had a significant influence on the estimate. Where possible, detailed discussions are held on special factors associated with a project that may not be properly handled in the present algorithm.

Figure 2 Relationship between productivity and productivity index for twenty-nine variables

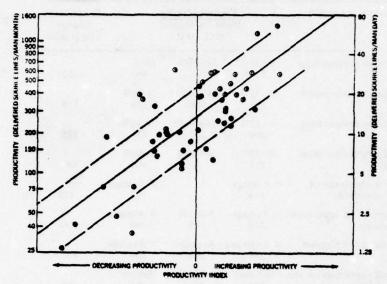
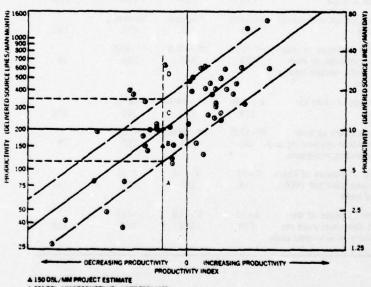


Figure 3 Estimated productivity for a hypothetical project



+ 200 DSL/NM PRODUCTIVITY INDEX ESTIMATE

Figure 3 is a plot similar to the one shown in Figure 2, which is presented for a hypothetical project. The productivity index is computed for the project from responses to the proposal questionnaire and yields the expected productivity to be attained, as determined by the measurements data base. In the case shown in Figure 3, the estimated productivity is seen to be two hundred

Table 7 Additional productivity related variables

Response Change (low to high percent)	Productivity Change* (percent)
0 to 100	705
50 to 100	215
0 to 100	205
	(low to high percent)  0 to 100  50 to 100

<sup>\*</sup>Based on least squares fit to data in Figure 4

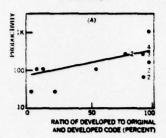
delivered source lines of code per man-month (DSL/MM), with one standard error range of 115 to 340 DSL/MM. The project team's independently developed productivity estimate for the same conditions was 150 DSL/MM. Thus, in this case, the project estimate is a more conservative estimate than that given by the productivity index.

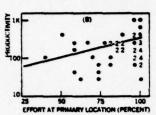
Consider additional conclusions that can be drawn from Figure 3. If we assume a normal distribution for the observations, when they are plotted as a log of the productivity versus a log of the productivity index in Figure 3, the probability  $P_A$  that the project would have a productivity estimate in region A (i.e., less than 2.06 or the  $\log_{10}$  of 115 DSL/MM) is about 0.17. The probability that the productivity estimate would be in region B (i.e., between 2.06 and 2.30, the log of 115 and 200 DSL/MM) is about 0.33. Similarly,  $P_C$  is 0.33 and  $P_D$  is 0.17. This is the probability distribution of productivity estimates, not the cumulative probability that a project will (or will not) achieve or exceed the productivity that was estimated.

Investigation is continuing into other variables from the data base that may also be related to productivity. Figure 4 shows several distributions that appear to have a significant relationship to productivity, although in two of these cases they are based on a limited number of observations. Table 7 expresses the net effect of the data plotted in Figure 4 in tabular form. Figure 4(A) shows productivity (in source lines of code per man month) plotted against the ratio of developed source code to the sum of any original (or reused) code plus the developed code. The plot in Figure 4(A) suggests that productivity is highest when there is no original or reused code, that is, when all the code is developed from the inception of the project. As the percentage of reused code grows, the expected productivity decreases.

other

Figure 4 Additional productivity relationships





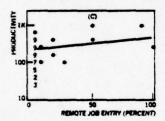


Figure 5 Relationship between decumentation and de

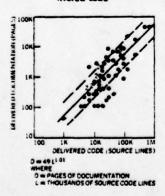


Figure 4(B), although it contains a large amount of scatter, suggests that when the development effort is spread across more than one location, i.e., as the percentage of effort at the primary location becomes less than 100 percent, the productivity decreases. Another question currently of interest is the impact of remote job entry on productivity. Most of the completed projects in the data base were developed without the use of terminals, as Figure 4(C) shows. On the basis of a least squares fit, however, those projects that use remote job entry do appear to have an increase in productivity.

## Other results of programming analysis

Although the primary effort has been directed toward productivity analysis, other analyses have been performed on the data base. Results of these efforts to the present time are presented here. The data can be used to check productivity estimates, and to check current project parameters against past experience, as reflected by the data base. Such results provide a multidimensional approach to crosschecking a number of the factors that enter into estimates of effort: productivity, duration, documentation, and computer costs. These results also indicate the nature of the analyses that can be performed against the data base.

Documentation is a critical product of every software project, and documentation costs are an important component of the estimation process. A useful parameter for measuring documentation is number of pages. Figure 5 is a plot of delivered documentation in number of pages versus delivered source lines of code. Documentation is defined here as program functional specifications and descriptions, users' guides, test specifications and results, flow charts, and program source listings that are delivered as part of the documentation. As a first approximation, the least squares fit indicates that a linear or first-order relationship exists: that is, the number of pages of delivered documentation varies directly as the number of lines of source code.

After programming project estimates have been completed, those estimates can be checked against the data base by using the plots in Figures 5-10. If, for example, the size of the delivered software product is estimated as ten thousand lines of source code (as shown in Figure 5) it can be seen from past experience that the expected number of pages of documentation to be delivered is five hundred. The range for one standard error for this given value is one hundred eighty to thirteen hundred pages. This provides an independent calibration point that the manager can use to compare his estimate against the experience of past projects. A significant difference between the two does

WALSTON AND FELIX

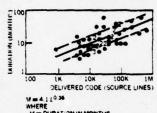
IBM SYST J

not necessarily imply an error on the part of the manager, but it does suggest that the assumptions and estimates might be reexamined.

The question of how much time to allow for the development of software is always difficult to assess. The relationship between duration (expressed in months) and delivered source lines of code is shown in Figure 6. Project duration as a function of total effort in man months is shown in Figure 7. Initial analysis indicates that a cubic relationship fits the data in both of these figures. This implies that the duration of effort increases by the cube root of the number of source lines of code delivered or by the cube root of the total effort applied to the development of the code. For example, for a project that is developing a software product of 10 thousand lines of source code, the expected duration of the effort is  $4.0 \times 10.0^{0.38}$  or 9.6 months. Figure 7 does not imply that simply reducing total effort automatically permits a reduction in project duration. Such a reduction would more likely make it impossible to produce and test the required volume of code.

project duration

project dure



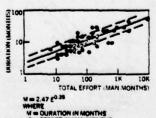
DURATION IN MONTHS

The staff size utilized to develop a given software product is influenced by a number of factors, including the time allowed for development, the amount of code to be developed, and the staffing rates that can be achieved. After a project has been estimated, one convenient measure used to describe the size of the project is the average number of people required. Figure 8 shows a relationship that can be used as another check on the estimating process. It shows the relationship between the staff size-expressed in terms of the average number of people (defined as total man-months of effort divided by the duration) - and the total effort applied.

size

Estimating computer costs is very difficult, but at the same time it can also be a very significant fraction of the total cost. Although only eighteen of the completed projects in the data base had computer costs reported, some interesting relationships are indicated when computer costs are compared with the amount of delivered code and the total effort, as is shown in Figures 9 and 10. In Figure 9, two observations (circled) are evidently out of bounds when plotted against delivered code. These same two observations, however, fit well with total effort, as shown by the plot in Figure 10. Based on this limited evidence, it appears that computer costs are closely related to effort, and they appear to have nearly a first power (or linear) relationship. Note that in Figure 9, the two out-of-bounds points are not included in determining the least-square fit.

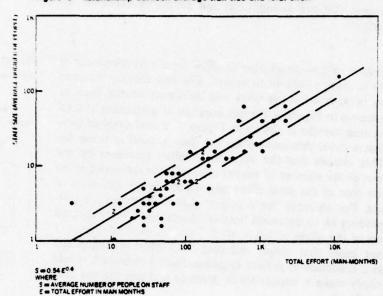
computer cost



NO. 1 - 1977

PROGRAMMING MEASUREMENT AND ESTIMATION

Figure 8 Relationship between average staff size and total effort

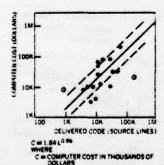


## Concluding remarks

# regression

The present approach to productivity estimation, although useful, is far from being optimized. Based on the results of the variable analysis described in this paper, and supplemented by the results of the continued investigation of additional variables related to productivity, an experimental regression model has been developed. Preliminary results indicate that the model reduces the scatter. Further work is being done to determine the potential of regression as an estimating tool, as well as to extend the analyses of the areas of computer usage, documentation volume, duration, and staffing.

Figure 9 Relationship between computer cost and de-



## **Appendix**

The effective utilization of programming measurements data requires the ability to store, retrieve, process, and report data. Specialized capabilities to do various types of statistical analyses are also required. These capabilities are provided by a Programming Project Measurement System. This system is composed of two subsystems, the question processing subsystem and the question analysis subsystem. The basic functions provided by the question processing subsystem are the maintenance of the data base (which contains the information submitted in response to the questionnaire), the retrieval and listing of data from the data base in various report formats, and the extraction

WALSTON AND FELIX

IBM SYST J

of data for transfer to the question analysis subsystem for statistical analysis. Figure 11 shows the overall flow of information in the programming project measurement system. The question analysis subsystem uses the Statistical Package for the Social. Sciences (a product of SPSS. Inc.), which is an integrated system of computer programs for the analysis of data, and provides the user with a large set of procedures for data selection, transformation, and file manipulation, and offers a large number of commonly used statistical routines.

Statistical routines include descriptive statistics, frequency distributions, cross tabulations, correlation, partial correlation, multiple regression, and factor analysis. The package has its own internal data management facilities that can be used to modify analysis files of data and can be used in conjunction with any of the statistical procedures. These facilities enable the user to generate variable transformations, recode variables, sample, select or weight specified cases, and add to or alter the data or the analysis files.

Project data enter the Programming Project Measurement System by way of questionnaires that are answered by project personnel. At the inception of the measurement program discussed in this paper, one questionnaire was used for both development and service (maintenance) contracts. On service contracts, questionnaires were to be submitted quarterly. For development projects, four questionnaires were to be prepared by the project at major milestones during the life of the project. Identical questionnaires were to be submitted, but not every item required an answer at each submission. The four reporting milestones were the following:

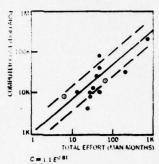
- Start of work.
- Preliminary design review or equivalent.
- Top-down programming completion of integration of onehalf of the program units, or Bottom-up programming-completion of unit test of three quarters of the program units.
- Acceptance test completion.

Problems arose when project personnel tried to use the same questionnaire form for both development and service contracts; differences between those two types of activities made it difficult to fit all the necessary questions into one questionnaire format. A further problem was the reporting frequency of development contracts. The four milestones might often be six months to two years apart, and many changes could occur in project organization, in project specifications, and in the definitions of products to be delivered, so that it was difficult to correlate questionnaire responses from milestone to milestone.

NO. 1 · 1977

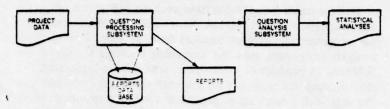
PROGRAMMING MEASUREMENT AND ESTIMATION

Relationship betw computer cost total effort



COMPUTER COST IN THOUSANDS OF E . TOTAL EFFORT IN MAN MONTHS

Figure 11 Programming project measurement system



For these reasons changes were made in the questionnaires and the frequency of reporting. Separate questionnaires were created for development projects (Software Development Reports) and for service efforts (Software Service Reports). Development reports, which cover detailed qualitative items as well as quantitative data, are submitted at the start of work and again at acceptance test completion. Between these two submittals, a Monthly Software Development Report is submitted. This is a one-page summary of the status of a product, cost, and effort that is submitted each month. The Software Service Report is an overview of a product that is being serviced and is submitted at the start and end of service. The Quarterly Software Service Report is a summary of the product, cost, and effort status, plus a detailed reporting of errors and their impact. Reporting is done by programming projects that are developing or servicing products in the form of lines of code and that employ two or more programmers with an expenditure of twelve or more manmonths of effort.

## CITED REFERENCES

- B. W. Boehm, "Software and its impact: a quantitative assessment." Datamation 19, No. 5, 48-59 (May 1973).
- F. T. Baker, "Chief programmer team management of production programming," IBM Systems Journal 11, No. 1, 56-73 (1972).
- HIPO-A Design Aid and Documentation Technique, Order No. GC20-1851. IBM Corporation. Data Processing Division. White Plains, New York 10504.
- F. M. Luppino and R. L. Smith, Programming Support Library Functional Requirements. U.S. Air Force, Headquarters, Rome Air Development Center, Griffis Air Force Base, New York (July 1974). See also Rome Air Development Center, Structured Programming Series, Vol. V.
- E. W. Dijkstra, "Notes on structured programming," pp. 1-82. O. J. Dahl.
   E. W. Dijkstra, and C. A. R. Hoare, Structured Programming, Academic Press, New York, New York (1972).
- F. T. Baker. "System quality through structured programming," AFIPS Conference Proceedings 41, Part 1, 339-343 (1972).
- M. E. Fagan. "Design and code inspections to reduce errors in program development." IBM Systems Journal 15, No. 3, 182-211 (1976).
- J. D. Aron. "Information systems in perspective," Computing Surveys 1, No. 4 (December 1969).

- G. F. Weinwurm and H. J. Zagorski. Research Into the Management of Computer Programming: A Transition Analysis of Cost Estimation Techniques. SDC Report TM-2712. System Development Corporation. Santa Monica, California (1965).
- E. A. Nelson, Research into the Management of Computer Programming: Some Characteristics of Programming Cost Data from Government and Industry, System Development Corporation, Santa Monica, California (November 1965).

#### . GENERAL REFERENCES

- 1. Improved Programming Technologies—An Overview, IBM Systems Reference Library, Order No. GC20-1850, IBM Corporation, Data Processing Division, White Plains, New York 10604.
- 2. P. Van Leer. "Top-down development using a program design language." IBM Systems Journal 15, No. 2, 155-170 (1976).
- 3. G. J. Myers, "Characteristics of composite design," *Datumation* 19, No. 9, 100-102 (September 1973).
- 4. W. P. Stevens, G. J. Myers, and I. L. Constantine, "Structured design." IBM Systems Journal 13, No. 2, 115-139 (1974).
- 5. E. W. Dijkstra, "Notes on Structured Programming," T. H. Report WSK-03. Second Edition, Technical University Eindhoven, The Netherlands (April 1970).
- 6. E. W. Dijkstra. "GOTO statement considered harmful." Communications of the ACM 11, No. 3, 147-148 (March 1968).
- 7. H. D. Mills. Mathematical Foundations for Structured Programming, FSC 72-6012, IBM Corporation, Gaithersburg, Maryland 20760 (February 1972).
- 8. H. D. Mills. Structured Programming. FSC 70-1070. IBM Corporation. Gaithersburg. Maryland 20760 (October 1970).
- 9. J. G. Rogers. "Structured programming for virtual storage systems." IBM Systems Journal 14, No. 4, 385-406 (1975).
- 10. G. J. Myers. Software Reliability: Principles and Practices. Wiley-Interscience. New York, to be published.
- 11. G. J. Myers. Reliable Software Through Composite Design, New York: Petrocelli/Charter (1975). Also see W. P. Stevens, G. J. Myers, and L. L. Constantine. "Structured Design." IBM Systems Journal 13, No. 2, 115-139 (1974)
- D. L. Parnas. "On the criteria to be used in decomposing systems into modules." Communications of the ACM 15, No. 12, 1053-1058 (1972).
- 13. N. Wirth. Systematic Programming: An Introduction. Prentice-Hall. Inc., Englewood Cliffs, New Jersey (1973).
- 14. J. F. Stay. "HIPO and integrated program design." IBM Systems Journal 15. No. 2, 143-154 (1976).
- M. F. Fagan. "Design and code inspections to reduce errors in program development." IBM Systems Journal 15, No. 3, 182-211 (1976).
- 16. G. J. Myers. "Composite design facilities of six programming languages." IBM Systems Journal 15, No. 3, 212-224 (1976).

Reprint Order No. G321-5045.

# ON THE RELATIONSHIPS BETWEEN DESIGN THEORY AND SOFTWARE LIFE CYCLE MANAGEMENT

Kenneth W. Kolence Institute for Software Engineering Palo Alto, California

(Presented at the Army Software Life Cycle Management Workshop 8/22/77)

ABSTRACT: A great number of the current problems in software product life cyle management can be traced to a lack of understanding of the design process in general, and specifically with respect to software. Because this workshop session is concerned with resource allocation and management, these topics are somewhat emphasized in the discussion that follows. However, the central themes are that the proper understanding of the design process should:

- (a) Significantly enhance the ability to break a design into parts which can be independently developed.
- (b) Significantly reduce the amount of both manpower and time required to complete a design, including implementation and installation.
- (c) Significantly enhance the ability of a design to maintain its integrity in the fact of functional changes, both during original design efforts, and subsequently during the product evolutionary phases.

#### \*\*\*\*

What is a "design?" What is the difference between a design and its realization? What makes one design "better than another?" What makes two designs similar? What is or are the central problem(s) of the design process? What is the "design process?" How do we partition a design into "man-sized chunks?" When does the design process start, and when does it stop? What, if any, differences exist between specifications and a design?

These and others are truly the basic questions we deal with when we look at the software life cycle and attempt to manage it. Yet, These questions have seldom if ever been directly faced, and so seldom if ever solved satisfactorily - certainly not theoretically, and only occasionally in practice. The theme of this paper is the need for a small and practical oriented research effort to answer

these questions in the context of the few basic philosophical approaches known in both software and normal engineering, and to define a set of experiments which can differentiate the characteristics of the processes developed.

Although design is perhaps the most characteristic activity of a technological civilization, it is perhaps the least understood and certainly the least recognized and researched of man's technical activities. Perhaps this is because we all do it - it is too normal of an activity to warrant serious attention. In those engineering arts and sciences based on the natural physics, traditions dating back two centuries give an appearance of understanding. And, these traditions do result in designs which work. Even here, however, no significant literature exists which directly addresses the basic questions posed above. In software engineering, the problem of design is more sharply in focus because we clearly do not have traditions that work. As in other areas between software engineering and the natural engineering, it is also probably true that the knowledge to be gained from an exploration and understanding of software designs and the design processes can significantly contribute to an understanding of the natural engineering design process.

The vein of practical value in an exploration of the idea of "what is a design," and "what is the design process" is unimaginably rich. The entire field of computer measurement and, more recently, software physics was a direct result of mining these ideas. Beyond that, clear relationships exist to many if not all of the fundamental problems that currently plague software engineering and life cycle management.

The starting point for the discussion is to make a clear distinction between a design and a realization of a design in some physical media. Take for example an actual automobile and a set of engineering drawings from which the specific automobile was constructed.

The set of engineering drawings is the design, and the specific automobile is one of the realizations of the design. Many realizations may exist: only one design exists. A second, and perhaps more illustrative example is the set of DNA chromosomes any animal or plant carries, and which is the determining pattern or design for that living thing. If we consider dogs, we know that many distinctly different breeds can be defined by their chromosomes, yet the essential "dogness" of each breed is not disturbed because inter-breeding is possible. Thus, we can consider dogs as having a design or pattern inherent in their chromosomes which defines "dogness," and sub-designs for different breeds which are like options and different models of the same automobile. This thought can be pushed much further, and results in the concept of "key structures" in a design. More on this point is considered later.

The important point in the above discussion is that a design is a pattern, from which a physical realization can be obtained. Clearly the existence of a DNA pattern represents a design in this sense. Equally clearly, DNA alone is insufficient to actually produce a living entity: an environment and set of processes (forces capable of using energy to cause state changes) must be available by which to cause some physical material to assume the states defined by the pattern or design. On reflection, this is true of any design which is to be used to "generate" one or more realizations. That is, a design not only specifies a set of states, but is only valid in the context of some set of processes. We shall call this set of processes the realization environment. For humans, the realization environment is provided in the womb. For cars, by a set of factories, mines, and power generating equipment.

It is most important to realize that for software, the realization form is a set of physical states in storage along with processors capable of changing these states by executing instructions. That is, the lines of code written on paper, and the data forms similarly

defined, represent the design, not the realization environment to be realized: this may be a compiler, assembler, loader, bookstrap loader, or other mechanism - software or hardware - but the design must be written in its context.

But if a line of code is to be considered as part of a design, then coding is part of the design process. What then of the rest of the process? And clearly, the code and data form specifications alone cannot be the only form in which a software design can be stated: "higher level" designs are known to be possible as well. Both engineering drawings and DNA provide examples which clarify this point, and which are equally valid. However, engineering drawing trees are more familiar and visual, so they will be used.

An engineering drawing tree is a set of drawings which can be visualized either as an upside down tree or a pyramid. At the top of the
pyramid is a single drawing: typically a three-view drawing of the
entire thing being designed. If an aircraft is being designed, it
shows the complete airplane. At the lowest level of design one has
"detail parts" or "purchased sub-assemblies." In the latter instance,
these effectively represent additional engineering drawing trees
that ultimately all end up at the "detail part level." In between
the master lines and dimensions drawing (at the apex of the pyramid)
lie many levels of drawings called "assembly drawings." The questions of concern are: (1) what does an assembly drawing represent,
and (2) what distinguishes a detail part drawing from an assembly
drawing.

A detail part is the lowest level of design, that is clear. But how does a designer know to identify the part as a detail part, rather than as an assembly drawing? The answer lies in the designer's knowledge of the realization environment. A drawing is called a detailed part drawing when a set of processes exist which are known to be capable of realizing the drawing design without further information.

A line of code in some language, with its operand data forms, will be realized in storage without further information being needed. It therefore is equivalent to the concept of a "detailed part" in engineering drawing terms. The general term I have defined for a "detailed part" level software design statement is an "operational relationship," since it may be a subroutine call (equivalent to a "purchased sub-assembly"), a complete sort program, or a source code statement in some language along with its operand data forms.

What is an "assembly drawing," which is typically made up of "subassembly" and detail parts drawings? Ultimately, of course, any design is defined in terms of detail part drawings when it is complated, but even then the assembly drawings are considered part of the design engineering drawing trees. An understanding of the answer is crucial to the entire concept of hierarchic design: assembly drawing represents an entity which is a complete design statement at the level of detail of the entity, but which must be further specified to be capable of realization. The assembly drawing shows the relationships of the parts, additional sub-assembly and/or detail part drawings are needed to specify the parts at a realizable level of detail. At a given level of detail, the design is completely specified by the corresponding assembly drawings even though the level of detail chosen may not be at the level of realization. Further, and of considerable importance to software design, each assembly drawing when decomposed will never (a) introduce new parts not visualized at the assembly level, nor (b) interface with other assemblies not visualized at the assembly level. Generally, neither condition is met by software designs represented by hierarchies of block diagrams and flow charts.

One of the basic flaws in the software design processes currently practiced is illuminated by the properties of engineering drawing trees, and particularly the assembly/sub-assembly/detail part discussion above. That is, levels of design statements are impossible

unless the design notation is such that (a) a design decision at one level can be detailed without introducing new interfaces, and (b) that more detailed design decisions can be mapped directly and uniquely back to a higher level decision. The problem is not simply a notational problem. The problem is fundamentally inherent in the view that software must be designed as a sequential set of operations. That is, it cannot be generally solved by any method which describes the design of the software as a sequential entity.

The reason for this general negation of a sequential view of software for design purposes can be seen directly by considering the "central question of design." It may be phrased in several different ways, each illuminating certain aspects of the problem. most general way, however, is simply "how can a structure or entity exhibit properties which are not inherent in any of its substructures." For example, how can an airplane fly when none of its parts can fly? Part of the answer to the central question lies in the basic description of the design process; i.e., the design process is the process of realizing overall properties by replacing functional requirements with structures and their relationships. More commonly, this is phrased as "replacing function with form," but this latter phrasing fails to explicitly mention that it is not simply structure which is being specified, but relationships between structures as well. It is precisely this point which cuts to the heart of the central design question, because some properties arise from relationships, and others are inherent to substructures and are transmitted to the higher level structures. Airplanes fly, the parts do not. The property of being able to fly arises from the shapes and other relationships of the parts. However, the mass of the airplane is simply the sum of the mass of each of its parts.

The central question of design has its implications on the software design process, and particularly on the method of representing a design notationally. In this context, the question is what structures to select to bring forth an entity with the desired functional

properties; i.e., that will interact with the presumed environment of any realization of the design in the desired way. Notationally, the problem is how to further represent it so that the appropriate realization environment can bring forth the actual realizations. In software, the language in which the design is specified to the detail part level (i.e., the operational relationship level) is, by definition, not usable at the higher and intermediate levels of design. And, of course, any truly general software design process must not only be independent of a specific language but even of a language type. 1 Therefore, it must be equally applicable to higher level procedural and RPG languages, as well as assembly code and even binary machine instructions. A multitude of possible realization environments must be acceptable for the same reasons. And, because a design needs to present all the information needed for a realization environment to bring forth a realization, the notation of software design must be able to link the "vagueness" of higher level design statements with the ultimate detail of the operational relationships of the design.

The process of selecting structures to bring forth desired overall (functional) properties needs to be examined in some detail at this point because of its rather considerable implications on the cost and time of design and realization. To begin with, the idea of the "key structure" of a design needs to be introduced. The "key structure" idea relates to the basic functional properties desired of the design, and in fact the basic key structure selected conditions the entire remaining design and therefore the design process. It also limits the feasible limits of design modifications and enhancements, both during the original design process and after realization.

The reason for this comes from completion of the "form follows function" dictum. The complete statement should be: "form follows function, function follows form, and the universe of possible designs is defined by the set of known structures and their properties." A specific language thus severely limits the sets of possible designs by its possible operational relationships.

A key structure is that choice of form to achieve the most basic functional property of the design realization. It is perhaps most easily explained by several examples. Returning to DNA, and therefore to the set of all species (both living and extinct), several choices exist for providing a permanent shape. A partial list includes shells, exoskeletons, and a vertebrate skeleton. Given a selection of one of these "key structures," the remainder of the "design" of the species becomes limited to a rather few basic overall forms. Mollusks all look like mollusks; lobsters, crayfish, and crabs are basically similar in shape, and vertebrates are instantly recognizable as vertebrates. The key structure is a basic design characteristic.

A second example from the set of entities designed by humans rather than evolution is perhaps more illuminating because one can visualize the thinking process that goes into the actual design process. sider the basic functional requirement for a device to travel from one physical location to some other location. The possible key structure choices here reflect things like wheeled vehicles (bicycles, automobiles, trains, etc.), flying things (balloons, gliders, powered aircraft), rockets, ground effect devices, ships, etc. Each key structure will accomplish the basic functional goal; i.e., has properties which interact with the presumed environment of the design utilization in a way that permits travel from one location to another. Each key structure, when selected, essentially defines the remainder of the design problem. And, all possible designs built around a given key structure can only have other properties within limited ranges. For example, the speed of a hot air balloon can never approximate the speed of a rocket. Nor can the cargo carrying ability of a rocket approximate that of a ship - at least not with currently known key structures to build a rocket around. So, the key structures for a given design can only be selected when all of the basic functional requirements are properly defined.

An understanding of the role of key structures and of the selection of a key structure from a knowledge of functional requirements is absolutely essential if changing functional requirements are to be effectively managed. The proper understanding permits one (a) to know how to specify functional requirements, (b) which types of functional requirements reflect fundamental design considerations, (c) what the impact of changed or additional functional requirements will be on the current design and the current state of the design process, and (d) what the implications are of each functional requirement in terms of structural (design) complexity and the cost and time required to design and realize the desired end item. Among additional benefits, the ability to partition a design into substructures with known interfaces is enhanced considerably, simplifying the design management task.

Again, an example is probably in order, especially relative to point (d) above. Assume a collection of functional software specifications, which includes one requirement which is relatively unimportant to the basic purpose for which the software is to be built. Assume this "unimportant" specification is that on-line updates to a data base can be made, even though it is almost always assumed that the data base will be updated "off-line." This "unimportant specification" will cause the "key structure" for the data base to be a significantly more complex structure than needed if only off-line updating is permitted. This added complexity will cause a step function increment in the costs and time to design, and to the operational "cost" in the sense of consumption of computing resources. Thus, it is essential to review functional specifications against their impact in requiring more costly key software structures.

Key structure considerations obviously are not limited to the original design process, but must be applied to the question of enhancements over time. A desired functional requirement applied to a design whose key structure cannot provide the properties can only result in essentially a new design. If it is "kludged" in

rather than redesigned, the complexity of the resultant code can easily lead to an unmaintainable system.

The key structure concept can be applied to any level of design, and is not limited to only the overall design. Therefore a hierarchy of key structure design choices appears during the actual design process. The major practical result of any key structure selection is to significantly constrain the design choices that are possible after the selection. As a result, as more and more of these decisions are made the possible choices of code become more and more limited. At the ultimate "detail part" or code level, little choice really remains and so the "coding activity" is normally treated as a very low level activity. This is true only if no key structure decisions are being made at coding time. the absence of an explicit understanding of key structures, coding often does involve some key structure decisions, and when this is so the result is usually added complexity and obscure code for maintenance purposes. At the least, it is a decision that needs to be carefully made in the full context of the functional requirements on the structure selected, not from the limited view of the coder at the time of selection.

I have chosen the term "design plateau" to reflect the impact of key structure decisions on design complexity and the concomitant cost/time implications. Visually, the complexity of a design remains relatively constant for a given set of higher level key structures, but shows a step function behavior between key structures. As an example, think of the design complexity of a child's wagon, a bicycle, an automobile, and a train. The design costs and time needed are clearly about the same for all possible wagons, for all possible bicycles, etc. But each "design family" has a significantly different level of cost/time associated with it.

A brief comment on the implication of key structure concepts on the "top-down" versus "bottom-up" design arguments is appropriate at this point. If a key structure is well known to the designer in terms of its structural requirements and functional behavior, then a pure "top-down" approach is possible. For example, if one is dealing with a purely sequential file structure, say for a log tape, then no surprises should be expected during the design process. The designer can see ahead sufficiently to the structure to use an hierarchic design approach and not get in trouble at the more detailed levels.

On the other hand, if the key structure properties are complex and especially if they are complex and the designer is not broadly experienced in their use, then a strong need exists to carefully detail the structure before design. Detailed coding may also be needed to explore the properties of the structure, and to establish how the desired functional properties are to be provided. In this instance, the "bottom-up" design approach appears to be needed. However, note that what is really happening is that the designer is exploring the implications of a high level design decision, so that the constraints generated by it are well known and can be provided for at more detailed design levels. This is the source of confusion between the two approaches, and an understanding of it clearly identifies when detailed code-level analyses are required.

The problem being addressed by the Software Life Cycle Management Workshop is how to manage the design process over the life of a software product. Implicit in the discussions is a view of the "design process" currently in use, and the simple fact is that the costs and time are as much a function of the design methodology as the design itself. This comes from the absence of an explicit understanding of how to partition designs (i.e., assign levels of key structure), how to notationally describe a design at high and intermediate levels in a directly substructurable form, and the relationship between functional requirements and structural complexity of a design.

In particular, management control and resource allocation over a software life cycle are directly related to a recognition of these problems. A satisfactory resolution of these problems will never be obtained in the absence of an explicity and formal understanding of the design process, and this in turn requires a similar understanding of the properties of a "design" in the sense at least similar to that considered here. The practical fact of the world today, however, is that these design concepts are not written up rigorously, nor have they been demonstrated in a sufficient number of instances to be used to teach the techniques. A "realization environment" does not exist yet for them. On the other hand, the Army has many projects currently underway and proposed for implementation in the coming year or so, and these must be done with current technical and managerial skills. So, it is clear that the workshop contributions are needed for the short term time frame (up to the next 3-5 years). But any basic set of solutions must, I believe, be based on a software engineering design methodology which is fully founded on a rigorous theory of designs.

Since the avowed basic purpose of this workshop is to define research projects to resolve the current life cycle management problems, I strongly recommend this research be funded and begun. In terms of money and effort, it should be kept small, because it is not amenable to "team research." The Army should, I suggest, pick a few people with a demonstrated ability to develop ideas of this scope and with practical experience, and "bet on them."

# BEYOND THE FOUR STAGES: WHAT NEXT?

AUGUST 22, 1977

David G. Robinson

D.P. MANAGEMENT CORPORATION
One Militia Drive
Lexington, Mass. 02173
617-862-8820

#### INTRODUCTION

The Stages Hypothesis of DP growth, (1) developed by Richard L. Nolan of D.P. Management Corporation, has proven to be a useful and reliable means of assessing the status of an organization's information processing activities. Originally based on three Harvard Business School research studies in the early 1970's, the Stages Hypothesis has since been verified in consulting studies conducted in over 40 organizations.

Since the original statement of the Stages Hypothesis in 1973, we have noted organizations whose DP growth and development has tracked well with Stages I-III, but which are diverting from the predicted movement to Stage IV. These developments have caused us to think further about the stages progression and to postulate the DP growth characteristics beyond the fourth stage. This paper presents some of our thinking in this area. It is organized into four sections:

- Overview of the Stages Hypothesis
- Recent Developments: Thoughts About the Fifth Stage and Stages V and VI.
- One Approach: Maintenance as the "Driver"
- "Repetitive Stages": An Alternative Hypothesis

<sup>(1)</sup> See Nolan, Richard L., "Managing the Computer Resource: A Stage Hypothesis." Communications of the ACM, Vol. 16, No. 7 (July, 1973), pp. 399-405.

COMPUTER SCIENCES CORP ARLINGTON VA

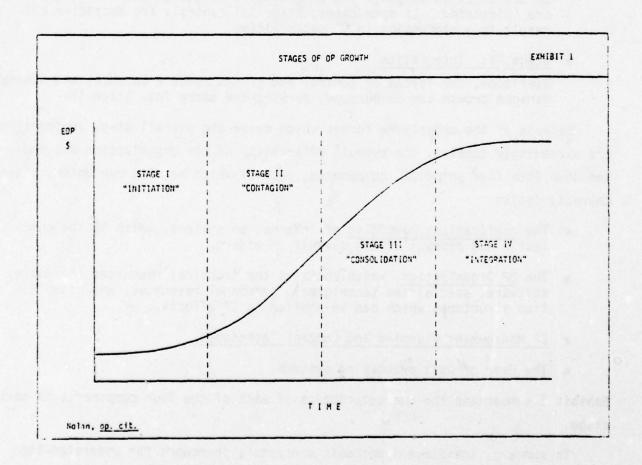
SOFTWARE PHENOMENOLOGY - WORKING PAPERS OF THE SOFTWARE LIFE CY--ETC(U).

AUG 77 B ELKINS, L HUNT

DAHC26-76-D-1006 AD-A053 014 UNCLASSIFIED NL 3 OF 8 ADA 053014 10 to 10 to

#### OVERVIEW OF THE STAGE HYPOTHESIS

The Stage Hypothesis was developed by Nolan out of an examination of the patterns of growth in data processing expenditures in three organizations. When total data processing expenditures were plotted against time, the resulting curve was "S-shaped." Nolan theorized that the S-shaped curve was a manifestation of underlying forces, and looked more closely at the characteristics of the study organizations at various points in time. This led to the hypothesis that the S-curve articulates the progression of an organization through four identifiable stages of DP growth, each stage representing a new phase in the organization's overall "learning" about information processing (Exhibit 1).



The four stages were defined as follows:

# • Stage I: Initiation

During Stage I the computer is introduced into the organization and basic computer technology is assimilated. After the initial expenditure for DP hardware and personnel, growth remains slow and steady.

# Stage II: Contagion

Stage II is marked by a dramatic upswing in expenditures and sustained rapid DP growth. New applications proliferate in all functional areas as the organization applies the technology initiated in Stage I.

# • Stage III: Consolidation

The rapid growth and uncoordinated proliferation of Stage II eventually results in growing concern on the part of the organization and its management about the effectiveness and efficiency of its DP activities. This is especially true since the rapid growth of Stage II is usually accompanied by weak or non-existent formal DP controls. As a result of the reaction to Stage II, DP growth is curtailed and formal controls are introduced. In some cases, Stage III controls are excessive and result in a regression in DP capabilities.

# Stage IV: Integration

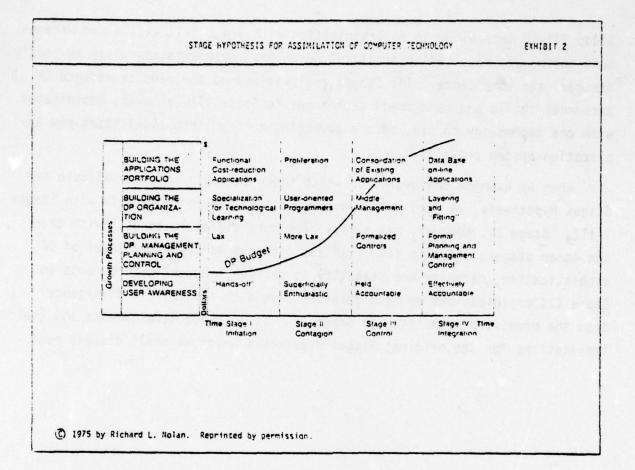
Over time, the forces of control and growth reach a balance, and steady, managed growth can be pursued, marking the entry into Stage IV.

Because of the underlying forces which drive the overall stage progression are exceedingly complex, the overall DP activity of the organization was broken down into four principal components, each of which has its own unique stage characteristics:

- The Applications Portfolio of information systems, which is the cumulative end product of the overall DP effort.
- The <u>DP Organization</u>, which contains the technical resources (hardware, software, specialized techniques), personnel resources, and organization structures which can be applied to DP efforts.
- DP Management Planning and Control Techniques
- The User of data processing systems.

Exhibit 2 summarizes the characteristics of each of the four components in each stage.

In summary, the Stage Hypothesis presents a framework for understanding the evolution of DP within an organization, and the way in which organizations learn about information processing.



# Further Work with the Stages

Since the development of the Stage Hypothesis in 1973, our work at D.P. Management Corporation has given us an opportunity to validate and further develop the Hypothesis in over 40 consulting applications to date in the public and private sectors. We have found that the DP growth pattern of these organizations has tracked exceedingly well with the Stages theory.

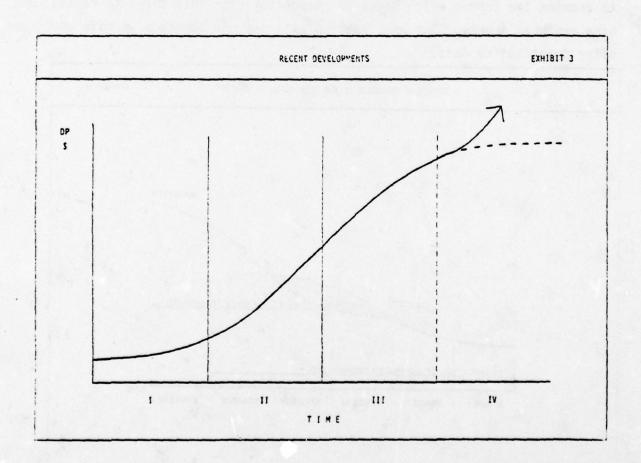
In our work, we use the Stages theory as the basis for a DP assessment technique we call a "Stage Audit." The Stage attributes shown on Exhibit 2 summarize "benchmarks" against which the particular characteristics of a given organization can be measured. This gives us a broad and well-integrated framework against which to view specific observed problems and formulate solutions. For example, one of our clients had undertaken a highly ambitious on-line data base program in an organization with inexperienced DP personnel and an almost total lack of DP controls and management techniques. This use of advanced

Stage III-IV technology in an organization with Stage I-II skills and management techniques resulted in a painful, overlong implementation which seriously damaged user confidence. The Stages analysis showed the need to advance DP personnel skills and management techniques to Stage III-IV levels coordinated with the technology in use, before advancing further into significant new application system projects.

When we examine the degree to which these in-depth studies validate the Stages Hypothesis, we find that our results track exceedingly well with Stages I-III. Stage IV, however, has always been more difficult to deal with than the other stages, since so few organizations have achieved the level of DP sophistication, maturity and stability it defines. Recent developments in Stage III organizations we have observed have also indicated a divergence from the predicted transition to Stage IV. It is these developments and their implications for the original Stages Hypothesis which we shall discuss next.

# RECENT DEVELOPMENTS: THOUGHTS ABOUT THE FIFTH STAGE AND STAGES V AND VI

Recently, we have noted organizations in Stage III which do not continue the stable growth pattern predicted for Stage IV. Instead, their expenditures have begun to rise markedly again, as they apparently enter a new phase characteristic of the earlier Stage II (Exhibit 3). Nolan first documented this phenomenon in 1975,  $^{(2)}$  and theorized that it was the result of the assimilation of major new technology. He presented three newly emerging technologies as

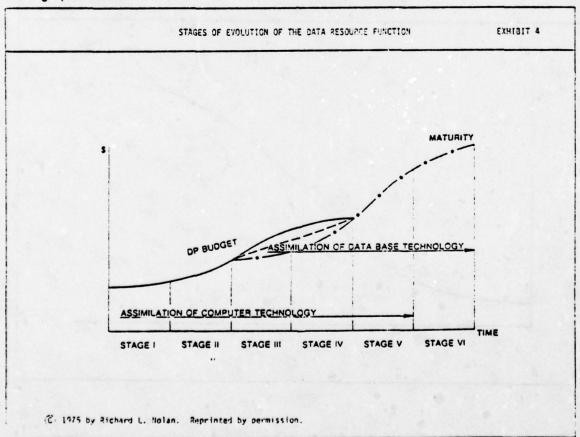


<sup>(2)</sup> Nolan, Richard L., "Thoughts About the Fifth Stage," <u>Data Base</u> (Fall-Winter 1975), pp. 1-12.

alternative explanatory hypotheses: data base management, minicomputers, and word processing. Data base technology appeared to be the central focus of this new growth resurgence. (3)

From that work, Nolan has recently postulated an extension of the original four stages to include Stages V and VI, as shown on Exhibit 4. (3) Here, the DP growth pattern originally experienced in the Stage I-IV progression describes the assimilation of computer technology into the organization; the progression of Stages III-VI is similar in character, but describes the assimilation of data base technology. Exhibit 5 postulates the specific characteristics of the Stage III-VI progression.

In conjunction with the development of Stages V and VI, we have attempted to examine the forces which might be responsible for this shift to rapid. data base centered growth. The next section outlines one approach we have developed using quantitative data.

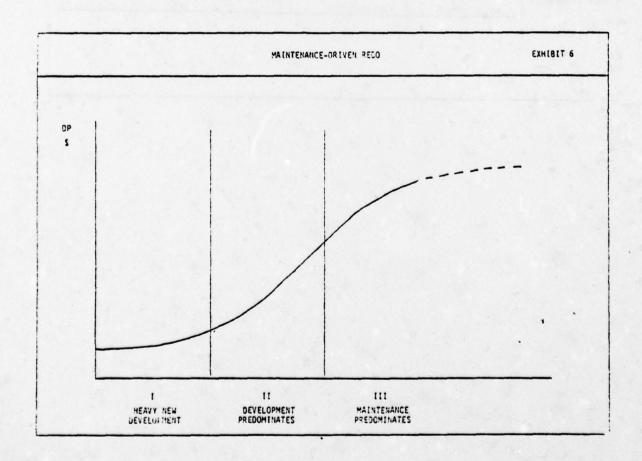


<sup>(3)</sup> Nolan, Richard L., "Restructuring the Data Processing Organization for Data Resource Management." IFIPS 1977 Conference Proceedings.

	i i		1	
PLICATIONS	CONSOLIDATION AND	DATA BASE	CONSOLIDATION OF	OPPORTUNISTIC
	BASE TECHNOLOGY		APPLICATION	
ORGANIZATION	MIDDLE MANAGEMENT	LAYERING	DATA	DATA RESOURCE ORGANIZATION
MCS	FORMALIZED - CONTROLS	TAILORED FORMAT	DATA STANDARDS	DATA-ORIENTED
ER AWARENESS	HELD ACCOUNTABLE	EFFECTIVELY ACCOUNTABLE	PARTNER	LEADER
	111	IA.	V	VI

ONE APPROACH: MAINTENANCE AS THE DRIVER

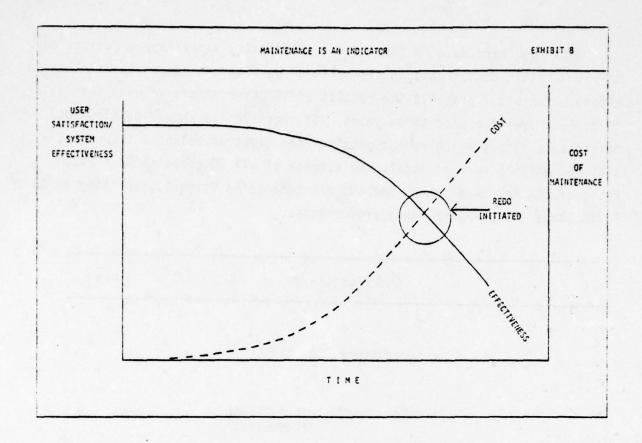
One of the measures we attempt to gather and analyze in our Stage Audit consulting studies is the ratio of new system development to maintenance across the Stages. Our benchmarks for this measure are shown in Exhibit 6. Stage I should normally be almost entirely new development. In Stage II, maintenance begins to appear, but new development still predominates. In Stage III, however, the balance shifts as rapid expansion is curtailed and the organization concentrates on maintaining the Stage II applications base to meet changing user requirements.



Obtaining hard data in this area is difficult, since many organizations do not maintain useful data on the ratio of development effort to maintenance effort. Exhibit 7 presents the results of 20 observations we have been able to collect over the past three years. Of these 20, 15 showed development/maintenance ratios which were typical of the expected balances shown on Exhibit 6, while 5 were atypical. An average of all 20 cases shows a rough ratio of 60% new development and 40% maintenance in Stage II, shifting to 30: 70 in Stage III as maintenance predominates.

STAGE MAINTENANCE DATA					
20 OBSERVATIONS, 15 "TYPICAL"					
• STAGE II AVERAGE: 42% MAINTENANCE 58% DEVELOPMENT					
STAGE III AVERAGE: 67% MAINTENANCE     33% DEVELOPMENT					

We view maintenance levels as an indicator of user satisfaction and system effectiveness. A given system, once developed, can be maintained to retain user satisfaction and effectiveness. But, as shown on Exhibit 8, there is a limit to the amount of periodic maintenance and enhancement which can be substituted for a complete system "re-do," as the effectiveness of maintenance progressively declines over time and its cost progressively increases. When the maintenance cost/benefit ratio is no longer positive, a complete system overhaul is initiated.



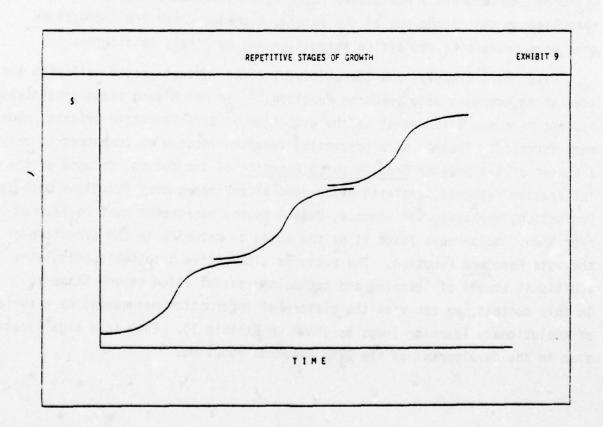
While this movement for complete system rewrites occurs throughout the Stages progression, it tends to concentrate in late Stage III, as the "baby boom" of Stage II systems mature at roughly the same time. The high maintenance ratios we observe in Stage III (Exhibit 7) are indicators of this overall tendency. Thus, during Stage III it is likely that a major overhaul of existing systems will be indicated. We see this occurring, with data base technology embraced as the vehicle for carrying it out. The broad scope of the data base program pushes the organization into the Stage III-VI learning pattern as it assimilates data base technology.

Nolan's current thinking sees Stage VI as maturity in the evolution of the management of the Data Resource Function. (4) As the author has examined the need to extend the original Stages Hypothesis, he has begun to consider an alternative to Nolan's six-stage hypothesis. This alternative hypothesis is presented in the following section.

<sup>(4)</sup> Ibid.

# "REPETITIVE STAGES": AN ALTERNATIVE EXPLANATION

The "Repetitive Stages" explanation views Stages V and VI not as just a one-time adjustment of the hypothesis, but rather a specific articulation of a more general theory. Exhibit 9 presents the author's idea of repetitive stages of growth, each of which represents the learning of a new information technology by the organization. The learning of computer technology thus occurs in Stages I-III; Stage III then becomes the base for the learning of data base technology in Stages III-V, and so on into new areas of information technology about which we can only speculate today.

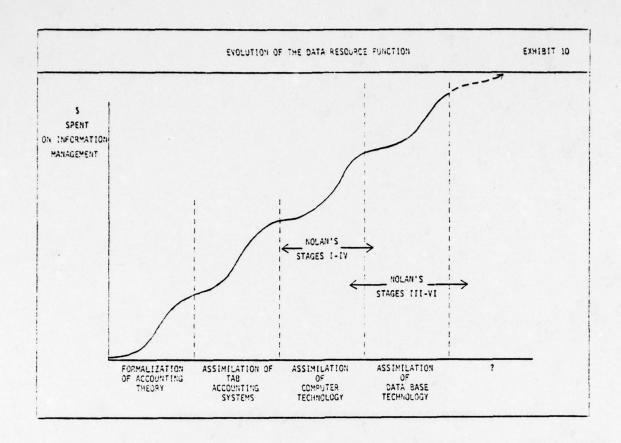


Using this explanation of currently observed events, the system "re-do" of Stage III is a repetitive phenomenon which occurs at different times for different organizations. For example, while some organizations we observe today are in Stage III, others are still in Stage II and will not reach Stage III for years. When an organization does reach the point in Stage III at which it perceives the need for a major systems overhaul, it will tend to assimilate the next major increment of information technology as the focus for the overall program.

Today, data base technology predominates as the vehicle we observe being selected. Yet the author theorizes that organizations which are today entering a new data base oriented growth period in Stages III-V will again emerge into a period of consolidation, only to eventually perceive the need for another major "re-do" in late Stage V. At this point, a new information technology would be embraced and a new stages learning progression entered. We can only speculate on the technology of the future, although there are indications that word processing and office automation may be likely candidates.

Taken more broadly, the "Repetitive Stages" concept tracks well with the idea of an emerging data resource function, (5) in which each stage progression represents a new advancement in the evolution of an information-oriented management function. Today's data processing function focused on computers is only a subset of the broader <u>Data Resource Function</u> of the future, focused on the information resource. As with other specialized management functions such as Production, Marketing and Finance, Data Resource management must be learned over time. Nolan sees Stage VI as the stage of maturity in the evolution of the Data Resource Function. The author's alternative hypothesis postulates additional stages of learning and technology assimilation beyond Stage VI. In this context, we can view the history of information management as a series of evolutionary learning jumps as shown on Exhibit 10. Each is a significant step in the development of the Data Resource Function.

<sup>(5)</sup> See Richard L. Nolan, Managing the Data Resource Function, (St. Paul, Minnesota: West Publishing Company, 1974).



# Our Next Steps

We have found the Stages Hypothesis to be a powerful tool in assisting organizations with the management of their information processing activities. We are currently updating our methodology continuously to deal with the conditions encountered in the Stage III-VI progression. Beyond this, the author intends to continue to examine his "Repetitive Stages" hypothesis as additional empirical data is gathered in our consulting work.

#### LIFE CYCLE PLANNING FOR A LARGE MIX OF COMMERCIAL SYSTEMS

by

I. R. Elliott
Systems & Programming Manager
Midland Bank Limited (U.K.)

#### 1. Introduction

Midland Bank has a centralised Systems & Programming Division with over 200 staff, maintaining and developing commercial systems for a variety of users within the Bank. Several Computer Centres in the U.K. are Due to our large investment in present systems we have a natural interest in preserving them for a reasonable time, and planning their We can claim from experience a working knowledge of the factors which compromise the life of a system, but our introduction to the more formal study of the subject has come from an association with Professor M. M. Lehman of Imperial College, London. Under his direction the evolution of a particular system was studied. It is the purpose of this paper to discuss the future of all our systems, with particular reference to the resources required to sustain them. strength of Systems & Programming in recent years is shown in Figure A (all figures are at the end of the paper). It will be seen the manpower has grown steadily and while we can excuse this in various ways, claiming a spate of original and renewed systems, we would not like to see the effort double again. The ageing of systems and the phenomena of low productivity and paralysis leading to "re-writes" has been masked in the past by other factors but will be much more significant in future with the complexity of systems now being supported.

## 2. Approach

A mixture of theory, or rather modelling, with observation of our actual situation has been used. As it is inherently difficult to quantify a

subject with so many variables, it is useful to check exploratory conclusions against a feel for an actual situation. The variety in both scope and characteristics of systems in the mix appears to be sufficient for generalisation of <u>our</u> situation but no claim is made that our model can be transported to another situation. Rather it is an approach or a way of thinking which may be of some use. Also our own thinking on program evolution is at such an early stage that we prefer to check with others before constructing a too shaky or parochial theoretical edifice.

The question we are asking is not so much "how long will systems last?" as "how long do we need to make systems last?". The life cycle of a system is subject to a large degree of management control, albeit drastic measures are needed in some cases. The influence of those responsible for development is of course just one of the many environmental pressures or constraints on the evolution of systems.

# 3. Definitions

We can for the time being permit ourselves a certain looseness about the definition of "system" as we shall address the characteristics of the total program source code as the most objective quantifiable factor of the situation. In considering the evolution of, for example, a mix of systems totalling one million lines of source code, this may be done at the "atomic" level of individual code lines or as (say) a thousand systems or sub-systems of one thousand lines each. In fact, an elementary model has been constructed and then the back-to-front approach has been taken of considering which useful situations are catered for by the model.

One problem of Life Cycle theory is the amount of introduction required before any progression is made, particularly as each stage of definition leads to reflection on its validity in practical terms. Let us first make the fairly obvious distinction between age and life times as applied

operationally, then its age may be measured up to a point of time such as the present. Its life is the time from birth to "death" when the code or system is discarded. Considering a system as an agglomeration of code, systems have a certain age since going "live" and a certain life before they are typically renewed. We shall generally consider code or system renewal. Our systems are part of the organisation we support and therefore every system becoming defunct must generally have a successor system (or sub-system, etc.) prepared. In fact, the renewal of an entire system is only a special case of the measures needed to adapt it to its environment, but one with major planning and resource implications.

## 4. Program Census

To provide a base of code and systems, a census was taken of all <u>live</u> code for which Systems & Programming was responsible on <u>l July, 1977</u>. The source code was classified by machine, language, and the dates it originally went live. For the remainder of this paper we consider amounts of source code lines irrespective of language used. There was a total of <u>l,918,000 source lines</u>. The average age of the code is 2 years 7 months.

The distribution of the code into its years of origin is shown in Figure B in histogram form. It is simpler for further extrapolation to present the data by measuring time from a fixed date in the past rather than relative to the date of the census. It should be noted that:-

- a. The origin of the horizontal (time) axis is 1 July, 1967. Amounts of code are shown in yearly intervals up to 1 July, 1977.
- b. The ten years shown will be referred to as "Year 1" up to "Year 10".

  Thus we still have, for example, 50,000 lines of code from Year 1

(produced between 1 July, 1967 and 1 July, 1968). 359,000 lines of new code went live in Year 10 (between 1 July, 1976 and 1 July, 1977).

c. It should be remembered that this is not a chart of the code we have produced over the past ten years, but that which has survived.

#### 5. Renewal Model

Suppose for the moment code will only last for a certain time before it is renewed, but that the total amount of live code remains constant. While it will be necessary to inject further elements of reality into the situation, this is a useful basic "steady state" situation. Whole systems at a time may be renewed, or we may be approximating the situation where small amounts of code are discarded and re-written. In this situation, the overall requirements of the organisation supported by the several systems are perhaps changing but of similar scope. Indeed if productivity in source lines was reasonably independent of language, then better languages could enable the organisation to accomplish its systems objectives with a constant amount of code. Having introduced this as a basic case, it is indeed tempting to try and make it happen. However, it is perhaps simpler to define the model than to be certain of the situations it describes.

- a. An array is used to describe distribution of code from Year 1 still in existence at a given time. We have generally initialised this with the amount of code (in thousands) from our real situation.
- c. Using the code and probability arrays, the model generates year by year the changing array of live code. Thus, starting in Year 11, for example, it would calculate for each of the previous years how much code should be renewed and add these amounts into Year 11 while subtracting them from the original years. Thus, at the end of Year 11 the code has moved on a little but its age must now of course be measured relative to the end of Year 11. The process is repeated for each year and ends after Year 20.

- d. The new code which has to be generated each year requires resources to be allocated ahead of time. From an amount of code a number of man years is deduced, then this is spread and accumulated over previous years.
- e. While they are variables of the model a rate of 3,000 source lines in one man year was used, and then effort was distributed during the two years prior to code going live. Approximately twice the effort is applied in the second compared with the first year. The "coding" rate includes every stage through feasibility study, analysis and design, specification, programming, system testing, etc.
- f. If this rate is applied to our existing base of code it shows an investment of 639 man years in our present code. This compares reasonably with the numbers (Figure A) in Systems & Programming when allowance is made for the code that has been discarded and the element of maintenance.

# 6. Simple Results

Nothing more is claimed for this method and theory than it should give guide-lines, and for intelligible results to emerge the number of variables has to be reduced.

We decided that for "average life m" we would use an approximately normal distribution with mean m and standard deviation m/6, in order to make it a function of one variable. This means that the only significant part of the distribution lies between m/2 and 3m/2.

Using the existing code base, we projected this for ten years trying average lives from three to ten years. In each case the array was printed out year by year and finally the "effort". The effort shows the number of men in a year required to produce the code being renewed, allowing for the spread already mentioned. Only the effort from

Years 11 to 18 is relevant since the model stops (arbitrarily) after Year 20. The "peak effort" is the greatest number of men required between Years 11 and 18, while "average" is the average over these years. These variables are plotted in Figure C against average life of code.

Looking at Figure C we must first remember this shows only one component of Systems & Programming manpower, devoted to the renewal of code. It ignores for the time being other aspects of maintenance, and the development of original systems which expand the total amount of code. Nevertheless it shows quite clearly that if systems last only three years, a force equivalent to the entire strength of Systems & Programming would be devoted to constant re-writes, an obviously untenable situation.

Looking at longer life times, the peak requirement does not decline against life quite as rapidly as the average because there is a greater spread in the life distribution as average life increases.

One must consider the deployment of other resources in Systems & Programming before evaluating the desirable average life of code, but my own feeling is that we must have an average life of seven years at the very least, preferably eight or nine, and ideally ten years.

#### 7. Expansion of Code Base

The code base used so far contains nearly two million lines originating over the ten years up to 1 July, 1977. Its renewal was considered while the total remained constant, but "new" code can be injected into the model year by year. In fact, the array input to the model of which the ten years up to "now" consists of existing code is extended by initialising it from Years 11 to 20 (in this case) with amounts of code giving a net increase in the total.

We modelled the situation where 100,000 extra lines of code are added each year to the code total. Thus, over a ten year period one million lines of code are added bringing the total base of code to nearly three million lines.

It should be emphasised that according to the average code life being modelled, the old and new code can be renewed more than once during the period being modelled.

As an example after the passage of ten years, i.e. at the end of Year 20, the code base is distributed as follows between Years 10 and 20 if the average life is eight years:

- 11, 24, 73, 165, 267, 355, 426, 451, 422, 373, 351
- e.g. at the end of Year 20, 73,000 lines remain which were written in Year 12.

The resource implications of this will be discussed in the next section.

#### 8. Maintenance

So far we have used the model to evaluate resources for what may be called the <u>dynamic</u> aspects of development - the renewal or regeneration of existing applications and the addition of further scope to systems, increasing the total amount of code. Before looking too closely at what this really means, let us have a look at the supposedly <u>static</u> aspects of maintenance. Of course, there is in practice every gradation between code with a low rate of change and complete renewal. The dynamic aspect of changing and renewing systems is taking place against a background of maintenance support. Obviously it costs more effort to modify 10 lines of code out of 100,000 than out of (say) 1,000 lines.

The "background" of code is always there because a system must remain operational until its successor takes over. Thus, the total amount of code that is live, and in practice expanding, requires at least a proportional effort to be added to the "dynamic" aspect already modelled.

What is the inevitable background effort on which to superimpose the "dynamic" aspects of system evolution?

To discuss this question let me first recklessly propound two "rules of thumb" for estimating the manpower required to maintain systems:

- a. Every 20,000 lines of existing code requires a person to support it.
- b. The number of systems staff required to maintain a system is one quarter of the number deployed in its final year of development.

On this basis the equivalent of 96 of my staff are looking after the 1,918,000 lines of code and another 50 will be needed after the amount of code has grown by another million.

Now, firstly the maintenance effort is not really linear with systems size, and secondly it depends on the dynamic activity. However, the linear rule has more validity for a mix of systems than an individual system. Obviously in practice our systems themselves interact, but up to a point it is surely reasonable to say that three million lines of code from various systems require half as much again background maintenance as two million lines.

Let us now summarise the manpower given by the model for ageing the original code with an average life of eight years, adding a million lines of expansion over ten years, and calculating background maintenance as the total expands.

Static (year end)	101	106	111	116	121	126	131	136	e
	171	190	212	236	259	272	271	264	<del></del>

This shows rather less people than we now have in practice (at the outset of Year 11). Further refinement would require a full discussion of the actual composition of staff (managers, support, etc.). It would also require a greater elaboration of any special factors in our present situation, not that there is anything especially peculiar.

Returning to the question of background maintenance, when it is said that one person is required for every existing 20,000 lines of code, this is to support a certain amount of change which has been resourced under the "dynamic" category. The formula can hardly be called accurate when a system which could quite literally be frozen would require no support. Put another way, we may say the linear formula for background maintenance includes a certain amount of "free" support and change of the system. Now in fact, as the amount of dynamic change increases against larger and larger static amounts of system, the interaction becomes much greater and we can no longer separate the two factors in the above table. You would need to add a third line, a sort of "increasing complexity" effect to our resources.

This question of the formula for resourcing rates of change of existing systems of various sizes opens up a subject in itself. Having scratched the surface, it is time to conclude with a few general remarks.

### 9. Conclusions

It is all very well to consider how those responsible for development may contain the situation within assigned resources, but the other factors at work make it difficult to achieve.

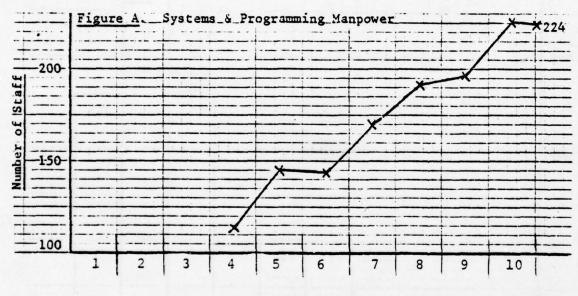
In the future we are working with a combination of rising user expectations and rapidly changing technology. If "Life Cycle Management" imposes constraints, may it not be seen as yet another reason why the systems people cannot do exactly what the user wants? While many people recognise intuitively that uncontrolled change compromises a system, the formal subject of evolution may appear so esoteric as to be merely the next set of excuses.

Were it not obvious for other reasons, it can also be seen that the exploitation of new technology must be a slow process according to the sort of theory we have discussed. This is really based on our experience of supporting systems in the past, and of course some

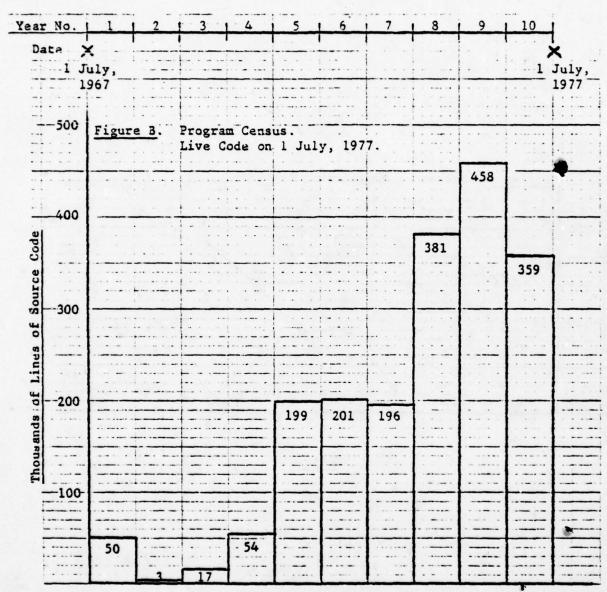
technological developments (further distribution, etc.) may enable organisations to break out from other restrictions apparently beginning to stall their progress. However, systems are systems and the effects cannot be escaped in the end.

A particular problem of present large systems is preserving continuity of operation through the renewal process. A form of metamorphosis must be achieved.

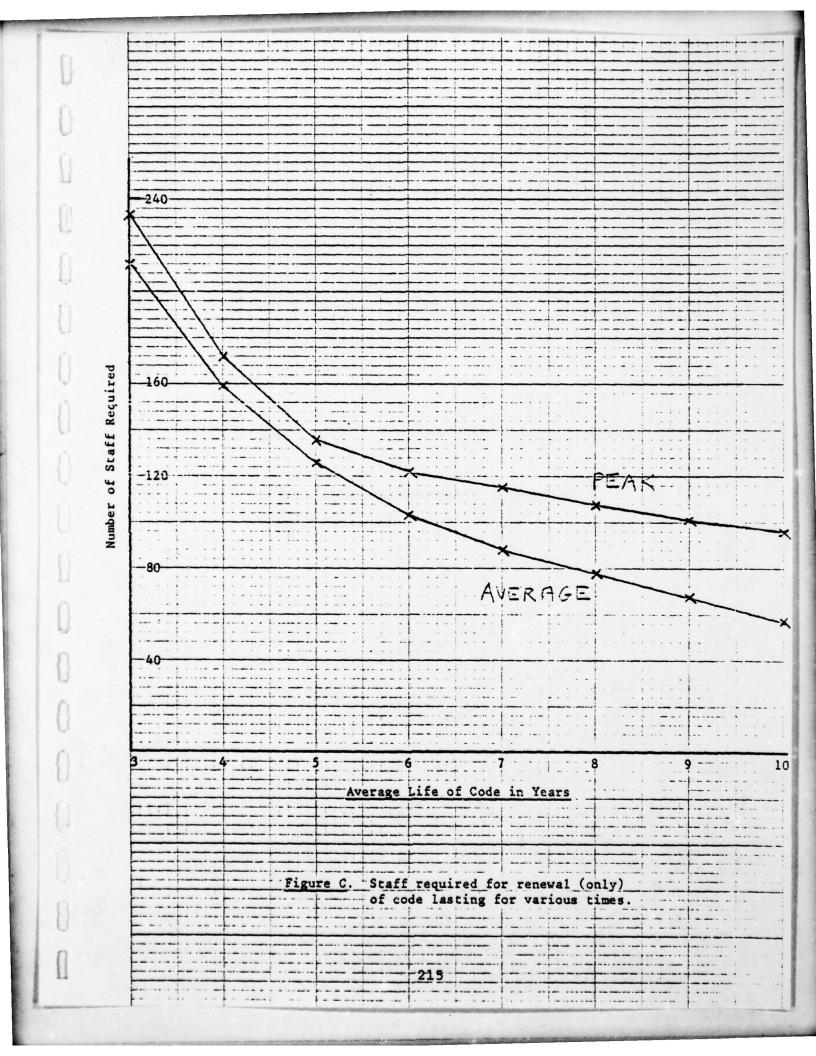
Finally, should we expect users to plan their requirements for a more specific time span? Should they anticipate the areas of flexibility they require over a desirable time span? Certainly practical planning is ineffective without the endorsement of the user.



Years



Year of Origin



### PROJECT LIFE CYCLE MODELLING:

BACKGROUND AND APPLICATION OF THE LIFE CYCLE CURVES

PETER V. NORDEN, Ph.D., P.E.

IBM CORPORATION AND COLUMBIA UNIVERSITY

The basic concepts and assumptions of a multi-stage model of resource cycles in Development Projects are reviewed. A few observations based on experience with this model are presented. These are: Reasonable stability of the cycles, early patterns in projects that led to success, and patterns observed when projects changed scope in midstream. A case study illustrating use of the Life Cycle Curves on a hardware development project is presented, and an Appendix showing how to compute the curves from simple Accounting data is given. The effects of changes in project environments, such as interactive programming and debugging, and other technological advances are liscussed, and the "single-cycle" possibility of software development projects is noted.

### PROJECT LIFE CYCLE MODELLING:

### BACKGROUND AND APPLICATION OF THE LIFE CYCLE CURVES

### INTRODUCTION

This paper consists of three parts:

- 1. Background and Principles of the Life Cycle Curves
- Application of the Life Cycle Model on a Complex Development Project: A Case History.
- 3. Appendices:
  - A) A Theoretical Model Explaining The Curves
  - B) A 'HOW TO' Manual for computing the curves.

    (This has tabular appendices of its own).

The intent of this structure is to present, in one place, a convenient compilation of enough material from earlier documents to allow the reader to:

- o Become acquainted with the use of the curves in practice
- o 'Walk through' a case study
- o Become acquainted with the underlying theory
- o Obtain a small reference manual for some do-it-yourself curve fitting.

In addition, comments on more recent experience, particularly with large software development projects, are provided. At the risk of some slight redundancy, the two sections and the appendices can be treated as stand-alone documents, and are paginated accordingly.

### BACKGROUND AND PRINCIPLES OF THE CURVES

The Life Cycle curves, more recently also called 'Rayleigh' curves, were discovered in the course of a study I conducted between 1956 and 1964 at the IBM Development Laboratories in Poughkeepsie, N.Y. The objective of the studies was to devise improved methods for estimating and mamaging large (hardware) development projects. The results were so interesting that we were able to persuade SCARDE (the Study Committee for the Analysis of Research, Development, and Engineering) of the R&D College of The Institute of Management Sciences, to allow me to gather project-pattern data from a large number of other companies. data were collected, using a blind-coding scheme administered by Professor A.H. Rubenstein of Northwestern University, which assured confidentiality of proprietary information. Cycles of the Rayleigh form were found in all these, again predominantly hardware, projects, and the analysis of the combined IBM and SCARDE data was published as my doctoral dissertation, and also as Reference (5).

The explanation of the underlying theory is found in Appendix A.

It uses a probabilistic model from reliability theory, originally due to Professor Benjamin Epstein, and describes the problemsolving behavior of scientists, engineers, and other technical professionals. It also provides the bridge from a model dealing

with probability density functions to the time series form in which manpower accounting data is conventionally kept, and in which it is displayed in 'life cycle' patterns. These density functions belong to the family of Weibull Distributions, and we communicated their applicability to development management, around 1959, to A. Pietrasanta, then with IBM's software development effort on the S/360 machine series. He, with P. Schreiber, early-on demonstrated their applicability to software projects. It is as a result of the above history, that we referred to the life cycle curves as 'Weibull' rather than 'Rayleigh' curves.

The curves were subsequently used with good results in several IBM locations, and in other organizations. Private communications informed us that among others, the then ESSO R&E organization found the cycle pattern to hold; and DOD used these findings to institute a procurement practice called 'Program Definition Phase' contracting. IBM, to this day, has retained a development protocol called 'Phase Reviews', which allows projects to be reviewed at each cycle completion point, and authorization to proceed, as well as funding, to be given by stages. Elsewhere in this collection of papers you will find, also, reports on the massive work and significant findings of Col. Lawrence H. Putnam, who applied the curves, and refined their analysis, at USACSC.

The principle of the curves is as follows: Research has indicated that there are regular patterns of manpower build-up and phase-out in complex projects. These patterns are made up of a small number

of successive phases, or cycles of work, throughout the life of the project. The cycles do not depend on the nature or work-content of the project, but relate to the way technical people tackle complex development problems. Each cycle can be described by the comparatively simple equation shown in the 'APPLICATION....' section, below, and yields a curve relating manpower used each month to elapsed calendar time in the cycle The single parameter (a) of the formula, governing the shape of the curves, can be thought of as a measure of 'crashiness' of the project: sharply peaked manpower duildups correspond to crash projects, while shallower curves are associated with more stretched-out programs.

Now, how do these cycles occur, and what do they mean? The best explanation is that engineering groups seem to work in definite surges of effort, and that each surge is associated with a particular purpose for doing the job.

In hardware development, the succession of purposes with which we work on projects generally is: planning, designing, building and testing a prototype, engineering activities associated with release of the product to a plant, occasional redesign, and a small number of cycles for product support and cost reduction. Figure 1 shows this sequence schematically. Sometimes it is preceded by an exploratory or 'concepts' cycle. In principle, this sequence is very widespread: Table 1 lists the cycle nomenclature established by several sources. The 'GUIDE' column refers to the computer users' group, GUIDE International Inc., whose

## TYPICAL WORK PHASES (CYCLES) AND ACTIVITIES IN ELECTRO-MECHANICAL DEVELOPMENT

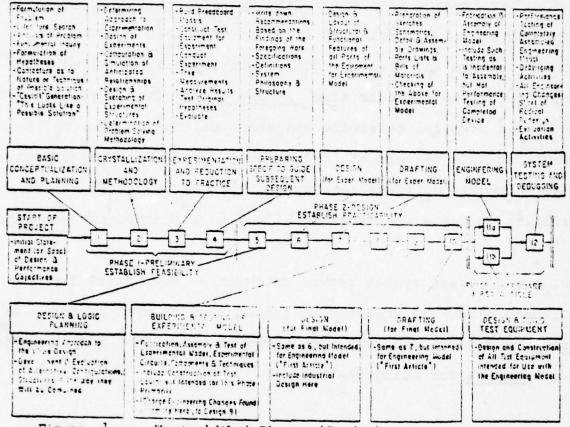


Figure 1. Typical Work Phases (Cycles) and Activities in Electro-Mechanical Development

## CYCLE (PHASE) NOMENCLATURE

G	-					ACE
ARON (SOFTWARE)	DEFINITION	DESIGN	IMPLEMENT		1.	MAINTENANCE
AR (SOI	DEF	DES	IMP		TEST	MAI
<b>3</b> 1		FURE	FIONS	NO		
BELADY (SOFTWARE)	PLANNING	ARCHITECTURE	SPECIFICATIONS DES. IMP'N.	INTEGRATION		SERVICE
ISO (SO	PLA	ARC	SPE	INI		SER
					z	
(V.)	<u>*</u>	ES'N.	IMPLEMENTATION		POST-EVALUATION	
GUIDE (SYST. DEV.)	FEASIBILITY	PRELIM. DES'N. DETAIL. DES'N.	LEMEN		ST-EVA	
(S)	FEA	PRI	IMP		POS	
(CSC)	z	IGN	EXTENSION	IF.	NT.	T.)
PE 0	PLAN	DESIGN	EXT	MODIF	MAINT.	(MGT.)
					PORT	
EN WARE)	ING	igts:	түре	SE	PRODUCT SUPPORT	
NORDEN (HARDWARE)	PLANNING	DESIGN	PROTOTYPE	RELEASE	PRODU	
				223		

Installation Guidelines and Standards Group is preparing a System Development Life Cycle Manual, from which the headings were taken.

The mathematical model of project manpower consists of the equation for each cycle, plus a linking function which specifies the relative sizes and durations of the cycles and their lags or spacing in calendar time. Typical inter-cycle ratios are given in Exhibit I, p.26, of Appendix B. The linking relationships have been encouragingly stable over a wide range of projects. This fact makes it possible to develop projections of manpower and time requirements for comparable projects, given a few actual points on the earliest cycle. In addition, early warning of significant departures from current schedules can be obtained by monitoring the deviation of actual manpower utilization from the established plan.

### Some Recent Experiences

There are three observations, arising out of recent experiences with life cycle applications, which are worthy of note:

Successful projects - these were generally associated with relatively high 'crashiness'. i.e. their cycles were of a pronounced 'sugarloaf' shape, and peaked early. Such a shape may be expected where you have an experienced project manager and/or project team, a high sense of urgency, and requisite resources readily at hand.

Piggy-back cycles - these are secondary 'bumps' on the projected primary cycles. They are generally associated with changes of project scope ( or re-engineering due to technical 'glitches'),

and when plotted in isolation (i.e. as residues from the primary cycle projection, on which they are superimposed) tend to have the shape of a standard Weibull or Rayleigh cycle. Such secondary cycles should not have their manpower included with the primary cycle on which they ride, when using the inter-cycle ratio projections, if they are due only to rework. Change-of-scope increments, however, legitimately add to the total amount of work to be done, and should be added in.

Software Development 'Single Cycle' Phenomenon - This is, perhaps, the most significant recent observation: Unlike hardware projects, in which the sum of the individual cycle curves <u>rarely</u> adds up to a 'pure' Weibull shape, software project development yields cycles that <u>generally do</u>. This would imply that software development is carried out, despite the fact that crude stages have been identified in it, as a tightly interlaced, functionally homogeneous effort. In Life-Cycle parlance, we could call it single-purpose. Perhaps this is due to the fact that, particularly with on-line coding and debugging, coding, testing, re-coding, re-testing, etc. are done at such a 'micro' level throughout the project, that the boundaries of macro-phases are lost and the whole job, de facto, is a homogeneous problem-solving effort.

One last observation may be worth adding. Table 2 shows the influence of modern programming productivity aids on the size and shape of project effort: The use, in this case, of Decision Table Processors has resulted in the reduction of total man-hours, in a controlled experiment involving a 492-mh project, by some 60%, and

	53%			47%				
DECISION TABLE APPROACH	20	=	22	20 }	\[ \( \tau_2 \)		100	
DECISION TA	40	21	44	39	54	1	198	
THOD	8	916	13	28 \ 230	35 \ 250	1	100	
CONVENTIONAL METHOD HOURS	40	80	62	136	174	1	492	
CYCLE	LEARNING	ANALYSIS	FLOWCHARTING	CODING	TESTING		TOTALS	

\* 60% REDUCTION IN TOTAL HOURS

TABLE 2

Spinoret a

a <u>shift</u> of the cycle peak to the <u>early</u> portion of the curve (From 37:63 to 53:47 percent). We have seen above, that the latter pattern is traditionally associated with successful projects.

APPLICATION OF THE LIFE-CYCLE MODEL ON A COMPLEX DEVELOPMENT PROJECT: A CASE HISTORY

### Introduction:

The Life-Cycle Model is an econometric model which describes the rate at which human effort is applied to the solution of complex problems. Since many development projects can be construed as an organized effort aimed at the solution of a group of related problems, the model can be shown to aid in forecasting the time and effort required to complete such projects.

The principal aim of this paper is to discuss a specific application of this model on a project in the Development Laboratory of IBM's former Data Systems Division. First, however, a brief description of the Life-Cycle Model itself.

### Life-Cycle Model: Theory of the Model

The Life-Cycle equation is:

$$y' = 2Kate^{-at^2}$$

where:

- y' = the manpower required in time period <u>t</u>. This is usually stated in quantitites related to the time period used as a base; for example, man-months per month.
- K = the total manpower required by the cycle, stated in the same units as y'.

- a = a parameter, defined by the point in time at which y' reaches its maximum value.
- t = time, in equal units such as weeks or months, measured from the start of the cycle.
- e = an irrational number 2,7128..., the base of the natural logarithms.

This equation produces curves such as those shown in Figure 1. The upper portion of this Figure demonstrates the shape of the curve when K is held constant, and a is allowed to vary. Using the curve with a equal to 0.0200 as a standard, an increase in the value of a to 0.0556 shifts the peak of the curve from the fifth month to the third month, and pulls the effective end of the curve from the eighteenth month back to the eleventh month. Conversely, a decrease in the value of a from 0.0200 to 0.0102 moves the peak to the seventh month and the effective end to the twenty-fourth month. Since these curves tail out to infinity, the effective end is considered to be that time period where manpower drops below one man-month.

The lower half of Figure 1 shows the effect of a change in the value of <u>K</u> when <u>a</u> is held constant. The shape of the curve, as determined by its peak and effective end, remains constant, but its height changes in proportion to the value of <u>K</u>. If <u>K</u> were set to 1, the result would be the dimensionless type of curve tabled in various handbooks.

The integral of the Life-Cycle equation is:

$$y = K \left( 1 - e^{-at^2} \right) \tag{2}$$

where:

y = the cumulative manpower used through month  $\underline{t}$ .

K = the total manpower required by the cycle.

a = a parameter determined by the time period in which the derivative of y reaches its maximum.

t = time in equal units counted from the start of the cycle

e = 2.7128, the base of the natural logarithms.

This integral equation can be rewritten thus:

$$e^{-at^2} = (K-y)/K \tag{3}$$

Then the Life-Cycle equation can be written in this form:

$$y' = 2at (K-y)$$
 (4)

This differential equation may be interpreted as follows.

The amount of manpower required in any given month depends upon the pace of the work (2at) and the amount of work left to be done (K-y).

There is a more fundamental explanation of this model, drawing upon the theory of random failures in life testing, but its exposition is too lengthy for this paper.

A development project consists of one or more of these cycles, each defining the manpower required to complete a particular phase of the project. Each cycle has the same functional form although each cycle will probably have different values of K and a.

The common cycles found in projects at IBM are:

Planning

Design

Prototype Assembly and Test

Release to Manufacturing.

Figure 2 is an idealized representation of these cycles, laid out on a time base as they might be in a real project. Some check-points are associated with various cycles to illustrate how forecasts of milestones can be made once their positions relative to the cycles has been determined.

### Fitting the Life-Cycle Model to Data

There are two methods of fitting Life-Cycle curves to data. The first assumes that the data is grouped by cycle. Then a least squares solution can be found for  $\underline{K}$  and  $\underline{a}$  by this formula:

$$\ln\left(\frac{y'}{2}\right) = \ln\left(2Ka\right) - at^2 \tag{5}$$

which is a rewritten form of the Life-Cycle equation.

When the peak of the curve is known, the parameters  $\underline{K}$  and  $\underline{a}$  can be estimated more readily:

$$a = 1/2t^2 \tag{6}$$

$$K = \left(e^{1/2} \cdot y'_{\text{max}} \cdot ty'_{\text{max}}\right) \tag{7}$$

When data is not grouped by cycle, it can be separated into its component cycles by the following procedure:

- Plot the data on a graph. It will be similar to the "Total Projected Man-Months Per Month" line in Figure 3.
- 2. At the point where the second cycle starts, there is usually a marked increase in manpower utilization, as can be seen readily in Figure 3.
- Fit a Life-Cycle curve to the data up to, but not including, this point.
- 4. Compute the values for the Life-Cycle curve.
- 5. Subtract these computed values from the observed data, and construct a new series composed of these residuals.
- Starting with the point where the marked increase was observed, repeat steps 1 through 5.

 Repeat this procedure until all cycles have been identified.

### Forecasting

Forecasts made by the Life-Cycle Model fall into two distinct types. The first is a forecast made before the project starts; the second is a forecast made on a going project.

Both types of forecasts require a knowledge of the various factors that relate successive cycles. These factors are:

- The ratio of the total time required in one cycle to the total time required in the next cycle.
- The ratio of the total manpower required in one cycle to the total manpower required in the next cycle.
- Where in one cycle the next cycle starts.

These factors must be determined individually for each installation. Those shown in Figure 4 have, however, proved useful as an initial set.

Using factors such as these, a Life-Cycle forecast can be made before the project starts by asking a responsible engineer to estimate:

- 1. When one of these cycles will start
- 2. The time required to reach his peak effort
- The manpower required at peak effort.

Using equations 6 and 7 aforementioned, the parameters of this cycle are computed. Then, using the factors shown in Figure 4, the parameters of all other cycles are computed. Finally, the manpower requirements by cycle are laid out in time.

The effectiveness of such projections can be gauged from an experience, not atypical. The project engineer had estimated a proposed project would require twenty-five man-years. Based on his answers to the questions above, Life-Cycle forecasted fifty-three man-years. The engineer raised his estimate, and the project started. On the basis of four months of actual data, Life-Cycle forecasted eighty man-years of effort. This happended to be the identical amount then forecasted by the project manager.

Once a project has started, a forecast is made by fitting

Life-Cycle curves to existing data, determining the number of cycles

which the project is likely to have, and forecasting these by means

of the relationships shown in Figure 3. The effectiveness of this

technique can be gauged by the case study which forms the second

half of this paper.

CASE STUDY OF AN APPLICATION OF THE LIFE-CYCLE MODEL

### Description of the Project

The objective of this project was to produce a system under contract to a customer to fill his particular need and at the same time to add a more general purpose version of this system to IBM's regular product line.

The system consisted of three separate boxes which operated in unison; i.e., no one box could process data usefully without the other two. One of these boxes contained a large mechanical device which was to be supplied by a subcontractor. The effort on this device was never included in the Life-Cycle projections.

The contract system was scheduled for shipment to the customer twenty-six months after the start of the project. The commercial system was scheduled for first shipment thirty-six months after the start of the project.

Organizationally, the project started in another Division of IBM but was transferred to the Data Systems Division after five months. The work was to be geographically relocated to Poughkeepsie, and an entirely new set of personnel assigned to the project.

Shortly after the Data Systems Division assumed responsibility for this project, the Operations Research Department interested the new project manager in the Life-Cycle Model and produced the projection shown in Figure 5.

Aside from the data just given on the project and its organization, the Life-Cycle analyst also knew that the Design Cycle had supposedly started in the fourth month of the project's life, that the replacement of manpower had started in the sixth month, and that the Prototype Assembly and Test Cycle was scheduled to start in the thirteenth month.

This projection indicated shipment of the first regular product line system about the time it was scheduled although the manpower forecast was substantially larger than the official Engineering Estimate anticipated. The project manager based his next budget request on this Life-Cycle forecast.

This first Life-Cycle forecast made no attempt to pinpoint shipment of the contract system.

The second Life-Cycle forecast on this project, shown in Figure 6, was made six months later. Several things had changed. The system was no longer expected to become part of the regular product line. The transfer of manpower had not occurred, and the project continued with its original manpower under the direction of the Data Systems Division. The project was encountering severe technical problems.

This second Life-Cycle forecast reflected these happenings in a larger initial Design Cycle and a secondary Design Cycle starting in the eleventh month of the project's life.

The third Life-Cycle forecast, shown in Figure 7, showed no significant change in the project's outlook. Shipment was now forecasted for the twenty-ninth month.

Figure 8 compares the manpower forecasts of the three Life-Cycle forecasts discussed in this paper. Even the first forecast fell within the generally accepted management limits of plus or minus ten per cent of the actual cost. The second and third forecasts were ideally close to the actual cost.

Figure 9 compares the shipment dates forecasted by these and two other intermediate forecasts. Again, all forecasts predicted well although the first forecast anticipated a system which was never produced.

In both time and manpower, Life-Cycle forecasted more accurately than conventional techniques used in the Laboratory.

The Operations Research Department learned several things from this project:

In forecasting the various cycles, it is best to use the <u>average</u> relationships between cycles, rather than to try to tailor these to a particular project.

The first forecast attempted to use factors based on one or two broadly similar projects.

Use of average relationships would have forecast manpower more accurately.

One must examine more detailed figures
 (i.e., type of personnel working)
 than the total manpower utilized per month when monitoring a forecast.

The second forecast might have been made two months earlier had this been done.

3. Even though the pattern of manpower
utilization might shift quite a bit in time,
the total manpower and time that the project
will require does not necessarily change by
a significant amount.

One implication of this is that the Life-Cycle

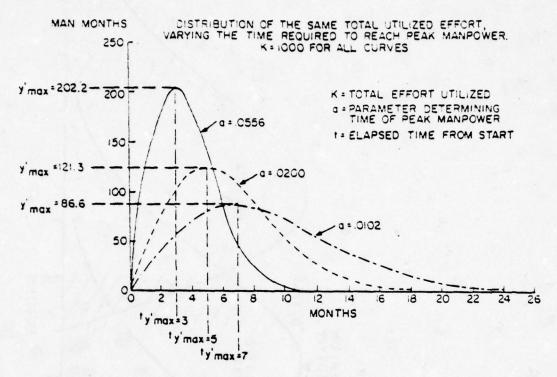
Model will do a better job of forecasting

manpower over a period than in any one

month.

In conclusion, the application of the Life-Cycle Model described in this paper proved the value of the technique to Laboratory management. It was subsequently used on larger projects where management regarded it as one of the numerous indicators which they consider when reviewing a project.

### MANPOWER UTILIZATION CURVE y'=2 Kate - at<sup>2</sup>



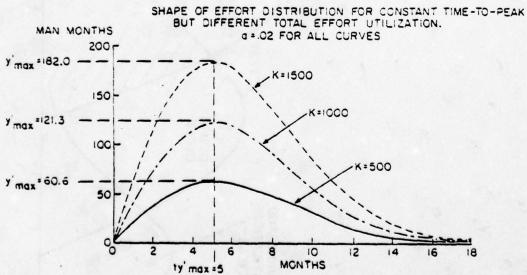
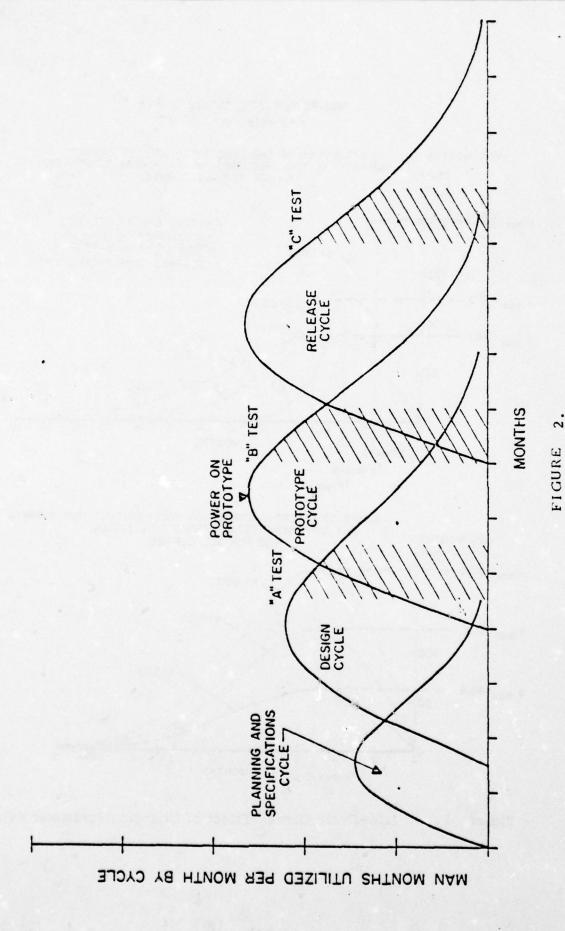


Figure 1. Life-Cycle Curve: Effect of Changing Parameter Values



[Telegraph

240

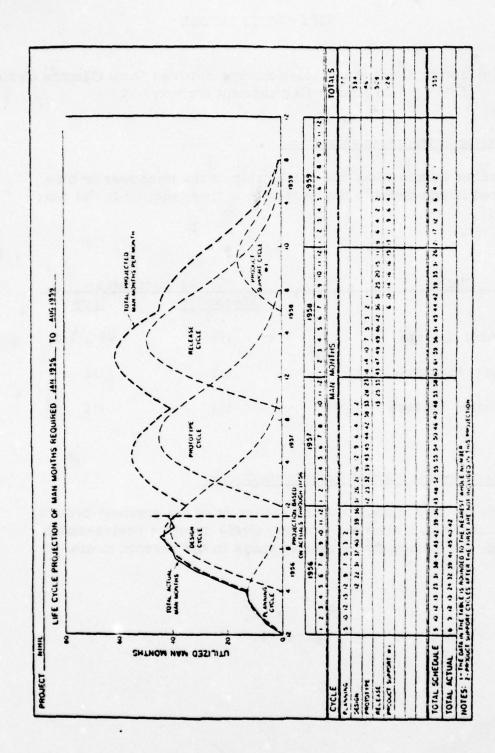


Figure 3 . Life-Cycle Projection of Hypothetical Project, Showing Composite Graphic and Tabular Report Form

### LIFE-CYCLE MODEL

Sample Factors Showing the Relationships Between Some Common Cycles
Found in Development Projects

### Manpower and Time Relationships

The tabled figures show the relationship of the manpower or time required in one cycle to the manpower or time required in the next cycle.

Cycles	Relationships			
	Manpower	Time		
Planning: Design	1:4	1:1 1/2		
Design: Prototype	1:1	1:1		
Prototype: Release	1:1	1:1		

### Lag Between the Start of Successive Cycles

A cycle usually starts about the mid-point of the previous cycle's time base; i.e., when the Planning Cycle covers a twelve-month period, the Design Cycle usually starts in the seventh month.

Figure 4.

PROJECTION MADE 7 MONTHS AFTER START

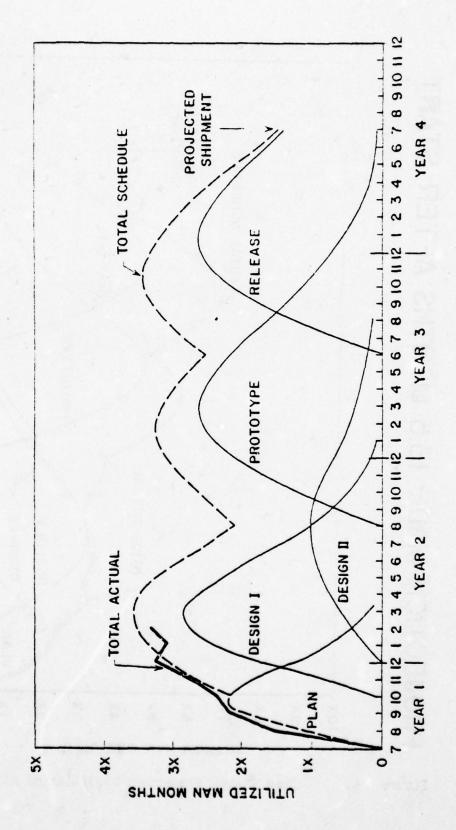


Figure 5. Case Study: Projection 7 Months After Start

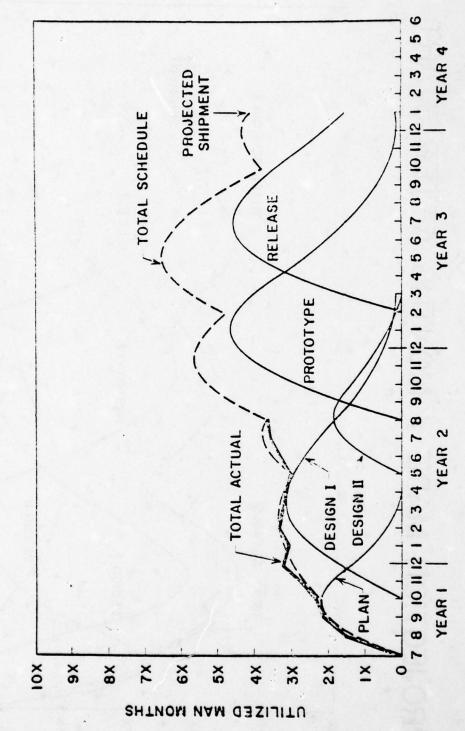


Figure 6. Case Study: Projection 13.5 Months After Start

PROJECTION MADE 20 MONTHS AFTER START

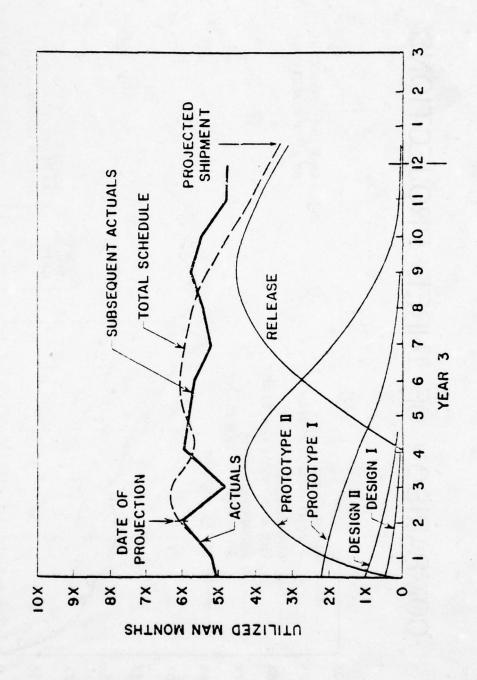
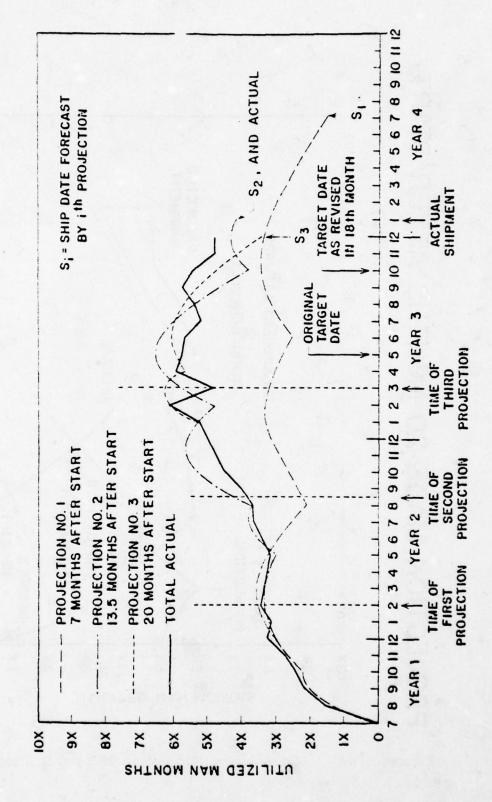


Figure 7. Case Study: Projection 20 Months After Start

# COMPARISON OF THREE PROJECTIONS



Control of the last

Figure 8. Case Study: Comparison of Three Projections

### COMPARISON OF ALL LIFE-CYCLE PROJECTIONS FOR MANPOWER

(Case Study Project)

### Projected Man-Months Per Year (Per Cent of Total Actual Manpower)

Year	Projection Months After			
	7	13.5	20	
1	9.3	9. 2	10.1*	
2	29.6	39.5	34.5*	
3	39.5	50.2	53.6	
4	13.9	1.3		
Total	92. 3	100.2	98.2	
*Actual				

Actual

### COMPARISON OF ALL LIFE-CYCLE PROJECTIONS FOR SHIPPING DATE

(Case Study Project)

	Time of Projection  Months After Start of Project				
	7	13.5	15.5	17	20
Projected Shipping Date*	36	30	31	31	29

<sup>\*</sup>Actually Shipped 30 Months After Start

Figure 9.

#### APPENDIK A

( Chapter references herein refer to Reference 5. )

The Engineering Design decisionmaking process discussed in Chapter V, leads to the following theoretical model. This can explain the observed patterns by asserting that an engineering development project can be considered as a set of unsolved problems.

This unsolved problem space is exhausted in the presence of human effort and creativity in a deliberate, purposeful group problem-solving setting. The operation of exhausting the problem space involves many activities which need not be ordered in time. There is, however, a partitioning of sets of these activities. This divides the unsolved problem space into a group of subspaces, which correspond to the purposes with which the problem-solving operation is concerned at various stages in the life of a project. In practice, the exhaustion operation is the design decisionmaking operation. In this context, the several problem subsets, each labeled by its "raison d'etre, " represent a stage-wise conversion of problems into solutions. If we make the following assumptions concerning each such subset, the effort utilization patterns actually observed can be constructed:

- The number of problems in the subset is finite, albeit unknown.
- 2. Human problem-solving effort constitutes an environment for the unsolved problem set and makes an impact

on the unsolved problems in the set.

- 3. Information gathering, gestation, identification of alternatives open for choice, and deliberation consume varying amounts of time. Any planning or design decision, made as a result of such deliberations, represents an event which causes one unsolved problem to be converted into a solved problem, thereby removing it from the unsolved problem space. If we assume the occurrence of these events to be independent and random, then the Poisson process would apply, and an exponential distribution of interevent times is a reasonable assertion. It is further possible to postulate a conditional probability function for the event that a problem will be permanently removed from the unsolved problem space, given that an identification event has occurred. Stated in another way, we postulate a conditional problem solution, given a certain distribution of insightful situations.
- 4. The parameter of the decision event distribution is a composite, implicit function of skill levels of problem solvers, level of exertion, administrative actions, and other random manifestations of the problem-solving situation, and interaction with the environment.
- 5. The number of people working in the engineering group at any given time is approximately proportional to the number of problems "ripe" for solution at that time. This leads to an interesting interpretation: The problem solvers act in a dichotomous manner as <u>catalysts</u> of the problem-solving process as much as <u>agents</u> who cause

problems to be solved.

The observed patterns can be shown to follow from the above assumptions. The model leads to a surge of effort, or cycle, several of which have been shown to comprise the life of the project.

Let us recall the work of D. L. Marples at Cambridge . He postulates that engineering design decisions have a tree-like structure, in which technical choices are made at each node. Subject to some technical criterion, one of a number of admissible alternatives is selected, symbolically representing a branch which leads to the next (problem) node.

Now consider a reliability model due to Epstein . He discusses a set of independent devices under test, subject to some environment E, which is a random process. Suppose this environment generates shocks, or "peaks" which are destructive to the devices in question. It is then reasonable to assert that such peaks (thermal shock, extreme vibrations, etc.) are distributed in a Poisson manner with some (rate) parameter. Let T be a random variable associated with the time interval between successive peaks. Then:

Pr(T > t) = Pr(No event occurs in the interval 0, t.) (1) where

t=0 is the time when the most recent event occurred. Then, from the poisson assumption:

$$Pr(T > t) = e^{-\lambda t}$$

(2)

and

$$Pr (T \leq t) = 1 - e^{-\lambda t} \qquad \lambda > 0, \qquad t > 0$$
 (3)

The p.d.f. associated with (3) is:

$$f(t) = \lambda e^{-\lambda t}$$
 (4)

This would describe the failure distribution in an all-or-none situation: Given a destructive peak has occurred, one device will fail with certainty. The number of items remaining, and the (average) number failing per time period, will also follow exponential functions.

Now suppose that the above situations were governed by a conditional probability of failure rule: Given that a peak has occurred in E, say that the probability of failure is some constant  $\underline{v}$ 

(0  $\leq$  p  $\leq$  1). It can then be shown that:

$$Pr (T \leq t) = 1 - e^{-p\lambda t}$$
 (5)

and

$$f(t) = \lambda p e^{-p\lambda t} \qquad t > 0$$
 (6)

This would result in a time-invariant conditional failure probability: Whenever a shock occurs, a device will fail with fixed probability. The items-remaining and number-failing-per-interval time series are again exponential curves.

Now consider the conditional probability of failure to be time

dependent. It can then be demo strated that if the conditional probability of failure is some function  $\underline{p(t)}$ :

$$Pr (T > t) = e^{-\lambda} \int_{0}^{t} p(t) dt$$
 (7)

whence:

$$Pr (T \leq t) = 1 - e^{-\lambda} \int_{0}^{t} p(T) dT$$
(8)

and:

$$f(t) = \lambda p(t) e^{-\lambda} \int_{0}^{t} p(t) dt$$
,  $t \ge 0$  (9)

This leads to the class of Weibull distributions, well known in reliability work. The physical interpretation here would be that the probability of devices succumbing to destructive shocks is changing with time.

In our case, it will be recalled, most of the data we have observed can be fitted by:

$$y = f(t) = 1 - e^{-at^2}$$
 (10)

which is a particular case of (3) when

$$p(t) = \alpha t, \tag{11}$$

where

$$a = \frac{\lambda \alpha}{2} . \tag{12}$$

This could imply that our engineers are learning to solve problems with an effectiveness which is increasing during each cycle. When one considers that familiarity with the problems at hand can lead to greater insight and sureness, this result is not implausible. Also, we see that parameter  $\underline{a}$  is compound: It consists of the insight-generation rate  $\lambda$ ,

and the solution-finding factor arphi . It should also be noted that equation (11) is a special case of the family of learning curves:

$$Y = ax^{b}$$
 (13)

which was discussed in Chapter II (Figures II+3, and 4). Substitution 21, 269 of (13) and (11) in (8) produces the general Weibull case .

The general, or 3-parameter, Weibull density function is given by:

$$f(x) = \frac{\beta}{\theta} (x - \alpha) \exp \left[ -\frac{(x - \alpha)^{\beta}}{\theta} \right] dx, \quad x \geqslant \alpha$$

$$= 0$$

$$x < \alpha \quad (14)$$

Here  $\alpha$  may be considered a location parameter,  $\beta$  a scale parameter, and  $\theta$  a shape parameter.

The general Weibull case is of interest in the present discussion because it allows explicit and general treatment of the built-in learning effect of (13). This can be seen most clearly by reference to a second approach of Epstein (op. cit.). He suggests a model based on the conditional probability of failure, or hazard rate g (t), where

$$g(t) = f(t) / \left[1 - F(t)\right]$$
 (15)

and shows that then

$$F(t) = 1 - \exp \left[ - \int_0^t g(\tau) d\tau \right]$$
$$= 1 - e^{-G(t)}$$
$$= \ge 0 \quad (16)$$

where 
$$G(t) = \int_0^t g(\tau) d\tau$$
, whence

$$f(t)$$
 =  $g(t) \exp \left[ - \int_0^t g(\tau) d\tau \right], \quad t \ge 0$ 

In our case,

$$g(t) = y'/1-y = 2 at$$
 (18)

which can be interpreted as a "survival ratio," or the probability that a problem will be solved ("die") in the next time period, given that it has survived (remained unsolved) thus far.

Epstein points out that the hazard rate model has much in common with the failure models. In fact, if the dependence of g(t) on the environment could be made explicit, one is led back to the earlier model, equation (8). The interpretation, here, that g(t), as p(t) in (13), is a time-dependent learning effect, is an attempt toward making this relation explicit. Following Epstein again, if

$$g(t) = kt^{k-1}, \text{ where } k > 0,$$

equation (16) becomes

$$F(t) = 0$$

$$= 1 - \exp \left[ -\left(\frac{t}{\theta}\right)^{k} \right], \qquad t \ge 0$$
(19)

and (17) becomes

$$f(t) = \frac{kt^{k-1}}{\theta^k} = \exp\left[-\left(\frac{t}{\theta}\right)^k\right], \qquad t \ge 0$$

$$= 0 \qquad \text{elsewhere} \qquad (20)$$

This is the Weibull distribution, and g(t) is

decreasing if 
$$0 < k < 1$$

constant if 
$$k = 1$$

increasing if 
$$k > 1$$
.

In the Life-Cycle curve, Epstein's

$$k = 2$$

and

$$\frac{1}{\theta^2} = a.$$

This is consistent with our observations that engineering group solution-choosing effectiveness (analogous to the hazard rate) is increasing. If the parameter  $\underline{b}$  in (13) were other than  $\underline{l}$  (which leads to the above k=2), more general solution-choosing rates would be admitted, and the maximum manpower point would be:

$$t_{o} = \sqrt{\frac{b-1}{ab}}$$
 (21)

while the points of inflection of the manpower cycle would become

$$t_1 = \sqrt{\frac{3 (b-1) \pm \sqrt{(b-1) (5b-1)}}{2 \text{ ab}}}$$
 (22)

These are the generalized equivalents of the values for (10), given in Chapter IV, Section F. Equation (22) behaves as follows:

- If b < 1, roots are imaginary.
- If b = 1, all roots are 0; trivial solution.
- If 1 < b < 2, we get one positive root or one point of inflection.
- If b = 2, we get 2 roots, one of which is positive, the other zero. (This is the Life-Cycle curve.)
- If b > 2, we get 2 points of inflection.

We decided to limit practical applications of the Life-Cycle method to the function in equation (10) because:

- 1. The general case is cumbersome computationally.
- 2. The general case requires estimation of (at least one) additional parameters, which cannot be justified by the amount and accuracy of data points commonly available in practice.

and

 The specific function chosen appears to produce projections adequate for managerial requirements. APPENDIX B

HOW TO MAKE

A LIFE-CYCLE ANALYSIS

Originally prepared by F.J. O'Reilly formerly with IBM, Poughkeepsie Dev't. Labs.

### DEFINITIONS: Cycle

A cycle is a stage in the life of a development engineering project, during which a major portion of the total required development work is completed. The project manager determines the number of cycles which will occur in his project, as well as the work content of each cycle, by the way in which he views and organizes his work. However, most projects are organized along the following lines (see Fig. 1):

# 1. Planning Cycle

- a. Define the technical objectives of the project.
- b. Outline the key technical problems, and investigate these in sufficient detail to show that a practical solution is possible.

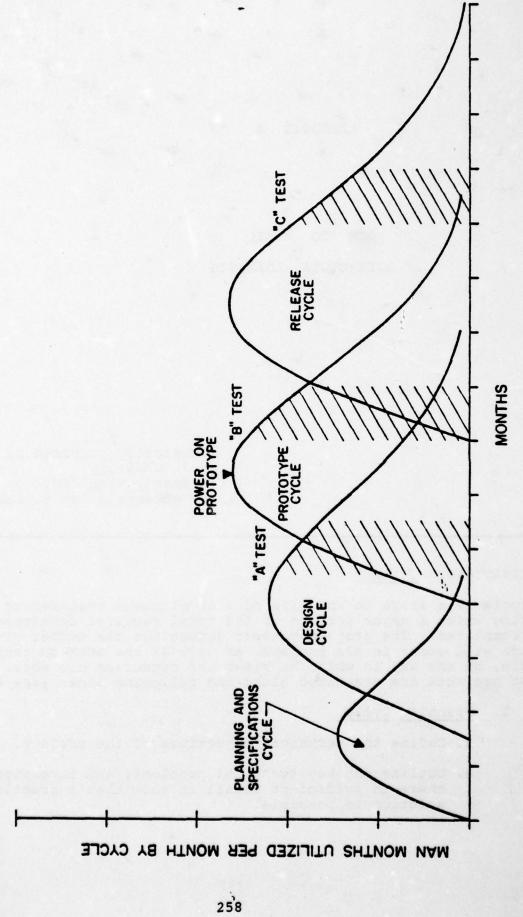


FIGURE 1.

- c. Establish specifications (i.e., design objectives) in sufficient detail to guide the detailed design which follows.
- d. Perform experimental hardware work necessary to achieve the aims of this cycle.

# 2. Design Cycle

All design, drafting, layout, experimental hardware, etc., necessary to produce the documents from which the prototype will be built.

# 3. Prototype Cycle

- a. Assembly and debugging of the prototype.
- b. Redesign and modification of existing design necessary to make the prototype perform in a manner which satisfies the project's objectives.
- c. Engineering test and evaluation.
- d. Preparation and release of engineering documents to manufacturing.

# 4. Release Cycle

All engineering laboratory activity on released documents necessary to

- a. Make the product more easily manufacturable.
- b. Correct deficiencies in original design.
- c. Update product's technology.

There are later cycles beyond the Release Cycle, but their content has not been defined for the purpose of Life-Cycle analysis at this writing.

The definitions are, of course, idealized. They are offered as a guide to the analyst rather than as hard and fast rules. Each situation must be weighed and interpreted on its own merits. Decisions about the content of the project and its cycles will always remain the first and most important job of the Life-Cycle analyst.

### III. MANPOWER UTILIZATION CURVE

## A. Introduction to the Curve

Tables of the manpower utilization curve have been computed. These appear in Appendix I.

The Life-Cycle Model describes each cycle within a project with the equation

$$y' = 2 Kate^{-at^2}$$

where

y' = man-months utilized in any given month

K = total man-months required to accomplish the work in the cycle

a = a coefficient which determines the month of peak manpower

t = time, counted in months from the start of the cycle

e = a constant 2.7183,.....

Figure 2 illustrates the mechanics of this equation. Line A, which represents the term 2Kat, increases by a constant amount each month. Curve B represents e-at<sup>2</sup>. Its value never becomes zero although it does become infinitesimally small. The combination of these two lines produces Curve C which is the manpower utilization curve.

This curve has many features which make it an ideal mathematical model. Some of these are valuable only to the theoretician. Four of them, however, are of practical value to the analyst:

1. The coefficient a determines the month in which manpower utilization is greatest. One can calculate this month by the formula

$$t_{y'_{max}} = \sqrt{\frac{1}{2a}}$$

Conversely, if one knows the month in which manpower utilization was, or will be, the greatest, he can calculate the coefficient a:

$$a = \frac{1}{2 \left( t_{y'_{max}} \right)^2}$$

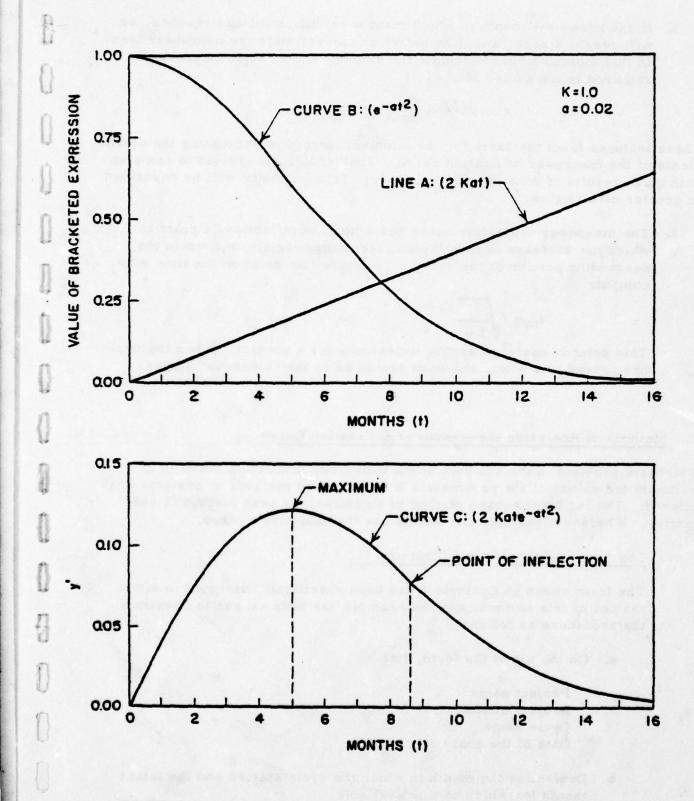


FIGURE 2. Some features of the manpower utilization curve.

3. If one knows the month in which manpower utilization has reached, or will reach, a peak, and if he knows or can estimate the manpower level in that month, he can calculate the value of K, the total manpower required in the cycle:

$$K = e^{1/2} (y'_{max}) (t_{y'_{max}})$$

These features form the basis for the simplest method of estimating the coefficients of the manpower utilization curve. They enable the analyst to come up with quick results of considerable accuracy. This technique will be described in greater detail below.

4. The manpower utilization curve has a point of inflection, a point at which the decrease in monthly utilized manpower slows down in the descending portion of the curve. To locate this point on the time axis, compute

$$t_{infl} = \sqrt{\frac{3}{2 a}}$$

This point is useful in setting milestones for a project. When the cycle has passed this point, the work should be in the "clean-up" stages.

# B. Methods of Analyzing the Manpower Utilization Curve

The basic problem in the analysis of the manpower utilization curve is to estimate the values of the parameters K and a. Two methods of analysis are offered. The log least square method is objective; the peak method is subjective. Wherever feasible, one should use the objective method.

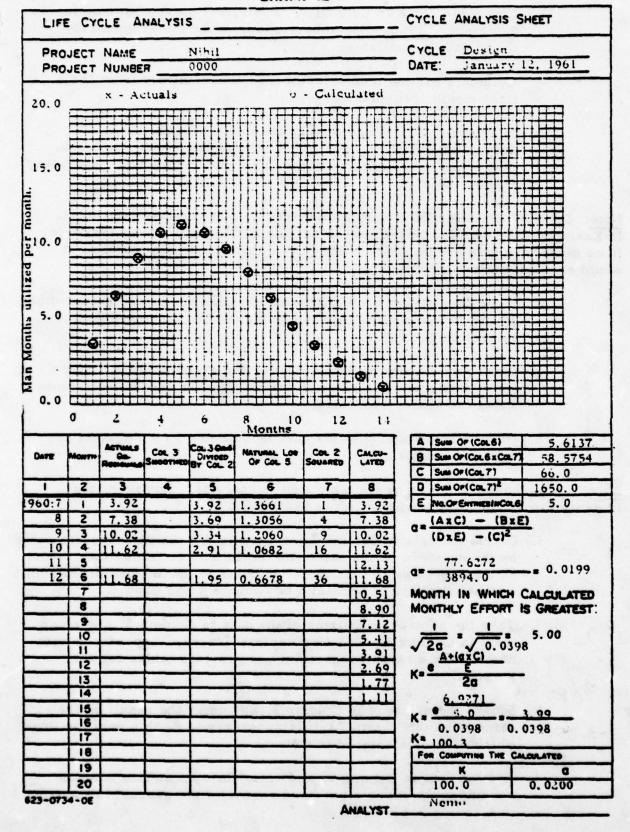
# 1. Log Least Square Method (Example 1)

The form shown in Example 1 has been specifically designed to aid in the use of this method, and the example has been chosen to illustrate the procedure as follows:

a. On the top of the form, list

Project name
Project number
Cycle name
Date of the analysis.

b. Determine the month in which the cycle started and the latest month for which data is available.



- c. List these dates in Column 1. Example 1's cycle started in September, 1960. December, 1960, is the latest month for which data is available.
- d. Column 2 has the months numbered consecutively from the first month, pre-printed.
- e. In Column 3 list the man-months utilized in the cycle. If data is not available for a given month, leave that line blank. Disregard the heading of this column until Section IV, Analyzing Project Data.

Note: DO NOT ELIMINATE A MONTH because data is not available for it. In Example 1 the data for November, 1960, the fifth month, is not available. If we moved the data for December, 1960, into the fifth month position, it would seriously affect the answer.

- f. If the data in Column 3 fluctuates erratically, it may be smoothed by any of several common methods, and the smoothed data entered in Column 4.
- g. Determine an appropriate scale for the graph at the top of the form, and enter this scale in the margins as shown in Example 1. Label the scales accordingly.
- h. Using small x's, plot the data in Columns 3 or 4 against the data in Columns 2.

Note: Graphing the data can be the first step in analysis. However, the interpretation of the graph will be left to Section IV.

- i. Divide the data in Column 3 (or Column 4 if the data was smoothed) by the numbers on the same line in Column 2. Enter the results on the same line in Column 5. Indicate in the heading of Column 5 whether the data in Column 3 or Column 4 was used.
- j. Take the natural logarithms of the data in Column 5, and enter them in Column 6. Use four decimal places. Appendix II tells how to take a natural logarithm and provides tables for this purpose.
- k. Square the numbers in Column 2, and enter the result on the same lines in Column 7. It is necessary to do this only for lines which have data entered in Column 6.
- 1. Add the data in Column 6, and enter the sum on line A in the box under the lower right-hand corner of the graph. In Example 1 this sum is 5.6137.

- m. Multiply the data on each line in Column 6 by the data on the same line in Column 7; add the products, and enter this sum on line B. No column is provided for the product of Column 6 and Column 7, since one can multiply and add to the result of a previous multiplication by using the accumulate-multiply feature on most calculators. Before starting this operation, it is advisable to clear and then lock the dials of the calculator so that the total cannot be removed accidently. In Example 1 this sum of the products is 58.5754.
- n. Add the data in Column 7, and enter the sum on Line C. In Example 1, this sum is 66.0.
- o. Square the data on each line in Column 7, add the results, and enter the total on line D. The calculating procedure is the same as that outlined in Step m. In Example 1, this sum is 1650.0.
- p. Count the entries in Column 6, and enter this count on line E. In Example 1 this count is 5.0.
- q. To calculate the value of the coefficient a, multiply line A by line C. Leave the result in the dials of the calculator.
- F. Multiply line B by line E, using the negative-multiply feature. Enter the result in the space provided. In Example 1 this figure is 77.6272. This figure must be positive.

Note: If the result is negative, check the positioning of the decimal point.

This positioning must be consistent throughout these operations. If the decimal point was positioned correctly, the difficulty is in the data. Stop the analysis at this point, and refer to Section IV for remedial procedures.

- s. Multiply line D by line E. Leave the result in the dials.
- t. Square line C, using the negative-multiply feature. Enter the result in the space provided. In Example 1 this figure is 3894.0.
- u. Divide the result of Step r by the result of Step s. Carry the answer to four decimal places. This is the value of the coefficient a. In Example 1 it is 0.0199.
- v. Double the value of the parameter a, and write this figure under the square root sign and as the denominator of the equation for K. In Example 1 this figure is 0.0398.

- w. Divide 1.0 by the value under the square root sign, and take the square root of the result to two decimal places. In Example 1 this result is 5.00. This is the month in which utilized manpower will be the greatest. Round this figure to the nearest quarter, e.g., 5.07 becomes 5.00; 5.47 becomes 5.50; 5.63 5.75; 5.88 becomes 6.00. Look up this rounded figure in Appendix I on the line labelled "Month of Maximum Manpower." The value of a, to be used in computing the curve, is located on the line directly beneath it. Copy this value into the block labelled "For Computing the Calculated."
- x. Multiply the coefficient a, as calculated originally (in Example 1, 0.0199), by line C. Add line A to this result, and enter the result in the space provided. In Example 1 this figure is 6.9271.
- Y. Enter line E in the denominator of this expression, and divide. The result of this division is a natural logarithm. Refer to Appendix II to take the anti-log of this number. Enter this anti-log as the numerator of the next fraction. In Example 1 the anti-log is 3.99.
- z. Divide this anti-log by the value of 2a, entered previously. The result is the coefficient K. Round this figure off to a whole number, and enter it in the box labelled "For Computing the Calculated." In Example 1, K equals 100.3, rounded off to 100.0.
  - (1) Refer to Appendix I, the same column as in Step w, and compute Column 8. Follow the directions given in the Appendix.
  - (2) Plot Column 8 against Column 2, using small circles.
  - (3) Initial or sign the form to show who made the analysis.

# 2. Peak Method

As explained in part A of this section, one can estimate the coefficients K and a if he can establish the month in which manpower utilization is at a maximum, as well as the level of this manpower. The peak method uses these estimates in a trial and error method of curve fitting. The form shown in Example 2 has been developed to aid in the use of this method.

a. Plot the man-months utilized against the corresponding month in which they were utilized. Use small x's as shown in Example 2.

DATE: JAN. 12, 1960 K=1.65 y't CALCULATED = 100.1 CYCLE: DESIGN 3.95 7.38 8.90 10.02 2.69 1.77 11.62 12.13 11.68 7.12 10.51 5.41 3.91 FOR CALCULATIONS
K = 100.0
a = 0.0200 ACTUAL 10.02 11.62 11.68 3.95 7.38 ty'max=5.0 y'max=12.13 ₩ 4 9= 2 9 4 0 0 EXAMPLE 2 0 O-CALCULATED 0 0 0 0 MONTHS 0 0 \*\* ACTUAL PROJECT NIHIL PROJECT NO. 0000 0 0 MAN-MONTHS UTILIZED PER MONTH 0.0 15.0

- b. If the data exhibits an obvious peak, note the month in which it occurs. This is the value of ty' max.
- c. Determine the man-months utilized at this peak. This is y' max.

Note: The peak may seem to occur between two plotted points. Then take ty'max as the mid-point between the two t values of the plotted points. Take y'max to be slightly greater than the y's observed. In example 2, ty'max taken as 5.0, y'max is 12.0.

- d. Multiply y'max by ty'max
- e. Multiply the result by 1.65. The result of this calculation is the value of K. Note it as indicated on the form.
- f. Look up the value of ty' max on the line labelled "Month of Maximum Manpower" in Appendix I.
- g. Calculate and plot the curve on the form on which the actuals are plotted, using Appendix I. Use small circles as in Example 2. Calculate the curve only up to the last point of actual data.
- h. Inspect the result. If the x's and circles do not follow each other closely enough, adjust the values of y'max and ty'max' and repeat Steps a through h.

Note: A change in K changes only the height of the curve. A change in a shifts the peak of the curve along the time axis.

 When a satisfactory fit has been obtained, copy the value of the coefficient a from the appropriate column in Appendix I. Calculate the entire curve.

The peak method is the easiest method of analysis to learn and apply. Its primary advantages are the ease and speed with which it can be used, and its ability to handle situations where an objective analysis is impossible. The accuracy of this method, however, depends mainly on the analyst's experience with the Life-Cycle model, as well as his familiarity with the project which he is analyzing.

## IV. ANALYZING PROJECT DATA

Section III described the mechanics of fitting the manpower utilization curve to the data of a cycle. The analyst's problem, however, is the interpretation of his data, not its mere mechanical analysis. This section will establish some ground rules to guide the collection and the interpretation of data.

## A. Gathering Data

The collection of data may seem to be a simple matter. In practice, however, it can account for 75 to 90 percent of the time required to complete an analysis. The following general rules should aid in the first steps of data collection:

- Determine the appropriate unit of time to use for the analysis-day, week, month, quarter, etc. The month is recommended for use in engineering development projects that last a year or more.
- 2. Determine the unit of effort. If the unit of time is the month, the unit of effort should be the man-month. If the unit of time is the week, the unit of effort should be the man-week. Keeping the unit of effort consistent with the unit of time avoids spurious fluctuations in the data, which conceivably could seriously affect the analysis. Dollars are not recommended as a unit of effort.
- 3. Define the project. Follow the ground rules set out in Section II.

  Never exclude a portion of the effort simply because it is funded or
  accounted for differently than the rest of the project.

To illustrate: Two identical models of a unit are built simultaneously. One is assembled and tested by Laboratory personnel who use Laboratory funds. The other is assembled and tested by Manufacturing personnel who use Manufacturing funds. The stated purpose of the work on the second unit is to accelerate manufacturing learning. However, the testing also provides a significant amount of feedback to the Laboratory group, which affects development. The work on the second unit should be figured into the data for analysis.

- 4. Determine what cycle or cycles the project is in. Use the definitions in Section II to make this decision.
- 5. Obtain all project numbers, shop-order numbers, etc., which describe the project, as well as any numbers used to identify a piece of work to be excluded from the analysis.
- The accounting department is the primary source of all data on a project's manpower utilization.

- 7. Life-Cycle analysis requires data to be summarized at two levels:
  - a. The man-months which the project has utilized, by cycle.
  - b. The man-months which the project has utilized, by detailed totals at the various levels of the account number. For this purpose the "Department Working" code may be considered part of the account number.

To illustrate: Consider the account-code structure of the Development Laboratory in 1961:

A listing of all charges to the department level of detail, as well as totals for each level above this is required.

The data of 7 (a) is the data used for curve fitting. The data of 7 (b) is used when necessary to explain excessive fluctuations.

Note: If it is impossible to organize data by cycle, one may take a <u>total</u> of the man-months utilized in <u>all</u> cycles in a given month and analyze this by a technique explained in part  $\overline{C}$  of this section.

8. If the data outlined in the previous step is published in a report form, the analyst should familiarize himself with the procedures followed in preparing the report to make sure he understands exactly what the figures represent.

Of the eight steps outlined above, Step 3 is the most likely to cause difficulty. If the analyst defines the project inaccurately, he must revise his data and start his analysis from the beginning.

# B. Inside-Out Analysis

Inside-out analysis is the preferred method of Life-Cycle analysis. To use this method, one must have project data identified by cycle. The first step is to ascertain that the data is valid and complete. The validity of the data can be tested by asking these questions.

1. Are the methods of recording data sensitive to changes in operations and cycles within the project?

- 2. Are the operations which were carried out (planning, design, assembly, test, etc.) consistent with the purpose of the cycle?
- 3. Based on the nature of their work, should the "departments working" charge this cycle?
- 4. Does the most recent data submitted reflect the current status of the project accurately?

If any charges are questionable, one should talk to the project administrator or project manager about them. These charges may want to be transferred to another cycle. An example might be charges by an Industrial Design group to a planning cycle. The completeness of the charges may be tested by asking these questions:

- 1. What other cycle reasonably might have been charged with the work of this cycle? Check the validity of the data in both cycles.
- 2. Is it possible that data rightfully belonging in this cycle was charged to a non-cyclical category? (Some activities, such as those which contribute to capitalizable equipment, may be segregated for administrative reasons.) Test equipment might be a case in point. At times test equipment construction is charged to a separate account, carrying no activity code. For purposes of analysis, however, this may want to be transferred into a cycle.
- 3. Is the data on certain operations complete? Parts fabrication is an example. The project orders a part through the Purchasing Department. The work may either be done in a Laboratory shop or sent to a vendor. In such a situation it is best to eliminate all charges from the Laboratory shop, since their contributions to the project are vulnerable to arbitrary make-or-buy decisions.

When the data is checked for validity and completeness, a log least square analysis may be started. When the data is plotted, it should be subjected to critical examination. Comparatively few points often suffice to show the general shape of the manpower utilization curve. (Figure 3a).

If another form emerges (Figure 3b), the data must be rechecked. The form (3b) results from the cycle starting in the second half of a month. Since the curve "assumes" that the effort represents the entire month, the shape of the curve is distorted. In this case the solution was to eliminate the first month and its data from the analysis and to take the second month and its data as the effective start of the cycle.

Figure 3c shows no discernible trend. In this case one must go through the log least square analysis and evaluate the result. This evaluation may indicate

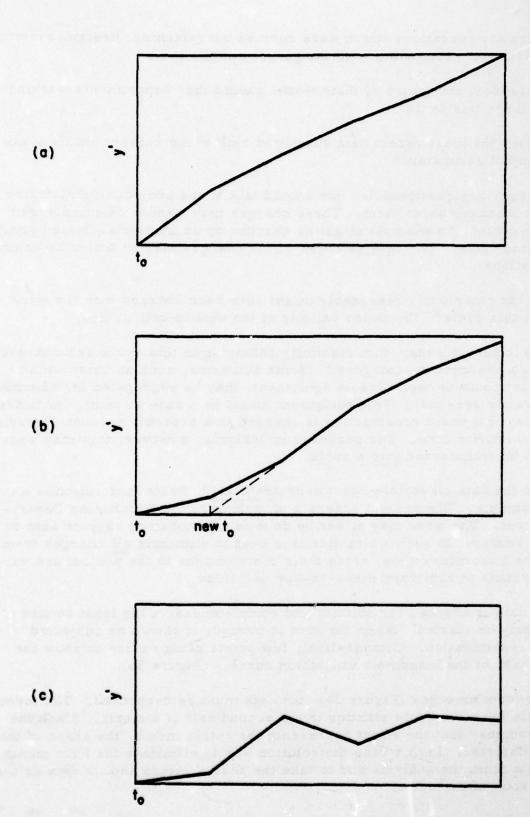


FIGURE 3. Examples of data one might find using inside-out analysis.

the need for a recheck and re-analysis of the data. On the other hand, it may show a cycle of greater duration than anticipated.

## C. Outside-In Analysis

One should use "outside-in" analysis to:

- 1. Analyze project data that is not organized or identified by cycle.
- 2. Double check interpretations based on an "inside-out" analysis.

The total man-months of project effort utilized in any month are the sum of the man-months utilized in each cycle in progress in that month. Laid out on a time scale, this sum is a curve which is the sum of the several cycle curves in the same period. Outside-in analysis uses the Life-Cycle model and the manpower utilization curve to decompose this sum curve, cycle by cycle, into individual cycle curves.

The first step in any Life-Cycle analysis is to check the validity and completeness of the data. Experience has shown that errors in the basic data are the primary source of difficulty in producing a Life-Cycle projection. Basically, the technique to use in this check is the same as that explained in part B above.

Example 3 has been worked out to illustrate the steps of this method.

- 1. Determine the cycle or cycles which the project is currently in. The project manager is the best source of this information.
- 2. Determine the number of cycles likely to exist in the data already.

  This information must also be sought directly from project personnel.
- 3. Plot the data (Example 3c). Look for a discontinuity in the plot which might indicate the start of the second cycle. This should occur approximately at the time the project manager indicated. If a cycle starts before the previous cycle has passed its peak, this discontinuity may not be too obvious. However, an idea of when the cycle should have started will help to pin-point the date.
- 4. Analyze the actuals from the first month to the month preceding the discontinuity, treating them as one cycle (Example 3d). Compute this cycle in its entirety.
- Subtract the cycle just calculated from the Total Actual to obtain a set of residuals. (Example 3b: Column 2 - Column 4 = Column 5).
   Do not compute the residuals for the data preceding the discontinuity.
- 6. Plot the residuals (Example 3e) computed.

-

### EXAMPLE 34

### DATA USED IN THIS EXAMPLE

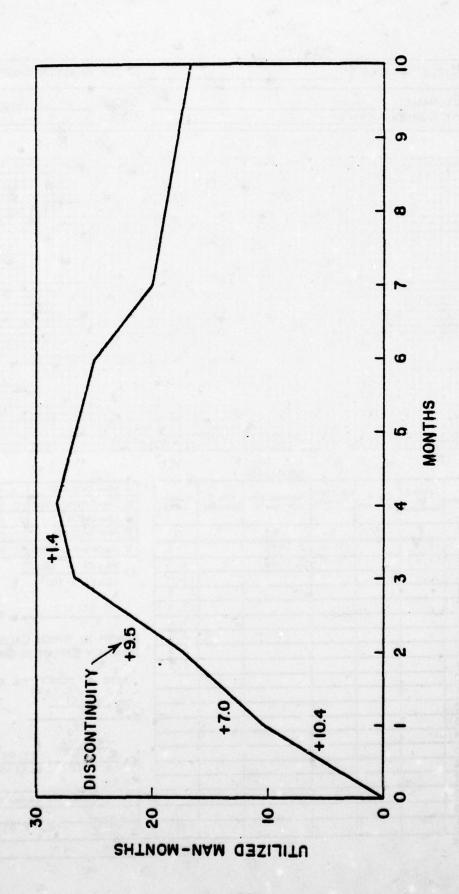
монтн	TOTAL	PLANNING	CTUALS BY CYC DESIGN	PROTOTY:	PE
1	10.4	10.4			
2	17.4	17.4			
3	26.9	20.2	6.7		
•	28.3	17.2	11.1		
5	26.6	13.4	13. 2		
6	25. 0	9.8	15, 2		
7	20.0	5. 0	15. 0		
	18.9	2.7	12.7	3,5	
9	18.0	1.0	8. 9	8.1	
10	16.9		6.7	10.2	
K		100.0	100.0	100.0	
•		0. 0556	0.0312	0. 0200	

### EXAMPLE 36

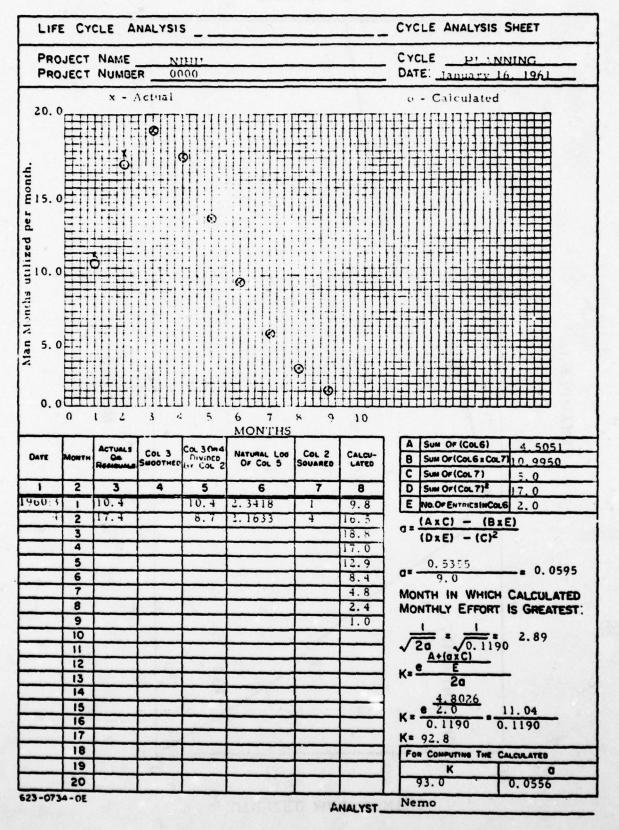
PROJEC	T: NIIIL					L	IFE CYCLE ANA	LYSIS
PROJEC	T NO: 0000	)				D	ATE: January 1	6. 1961
1	TOTAL	3 TOTAL		5	6	7	8.0	
MONTH	ACTUAL	CALCULATED	PLANNING	RESIDUALS	DESIGN	RESIDUALS	PROTOTYPE	
1	10 4	9. 8	9. 8				ent ad mark	
2	17.4	16.5	16.5					
•	26. 9	25.8	18.8	8.1	7.0			
4	28. 3	29.6	17.0	11.3	12.6			
5	26.6	28.7	12 9	13.7	15.8			
6	25.0	24.8	8.4	16.6	16.4			
7	20.0	19.7	4.8	15. 2	14.9			
	18. 9	18.8	2.4	16.5	12.1	4.4	4.3	
9	18.0	18.0	1.0	17.0	8.9	8.1	8.1	
10	16. 9	16.7		16. 9	6.0	10.9	10.7	



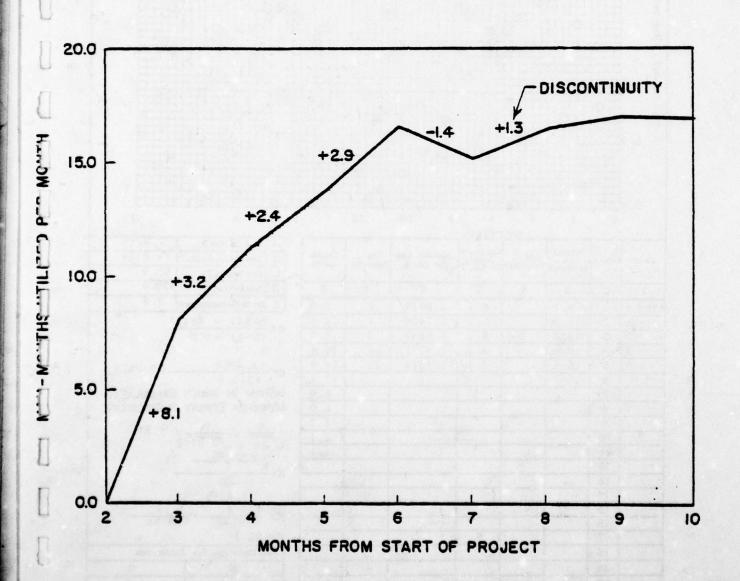
EXAMPLE 3c.



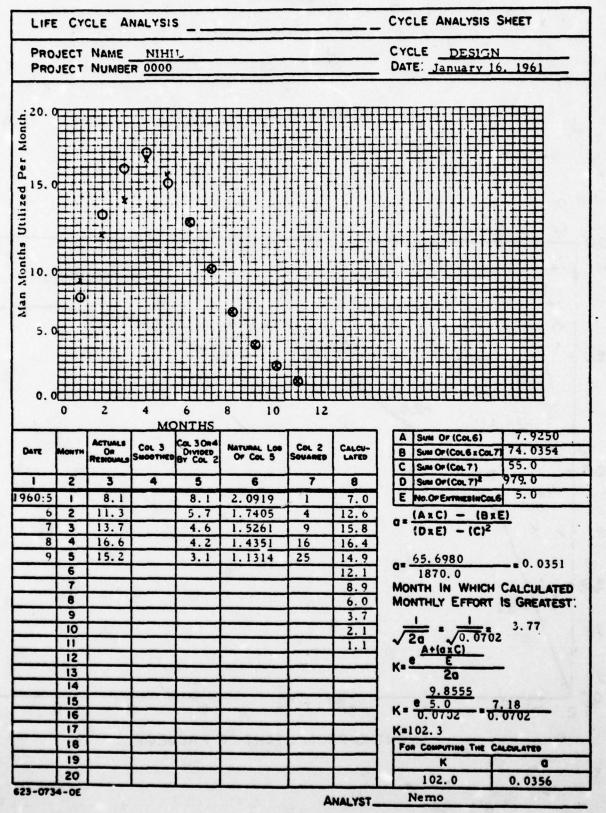
#### EXAMPLE 3d

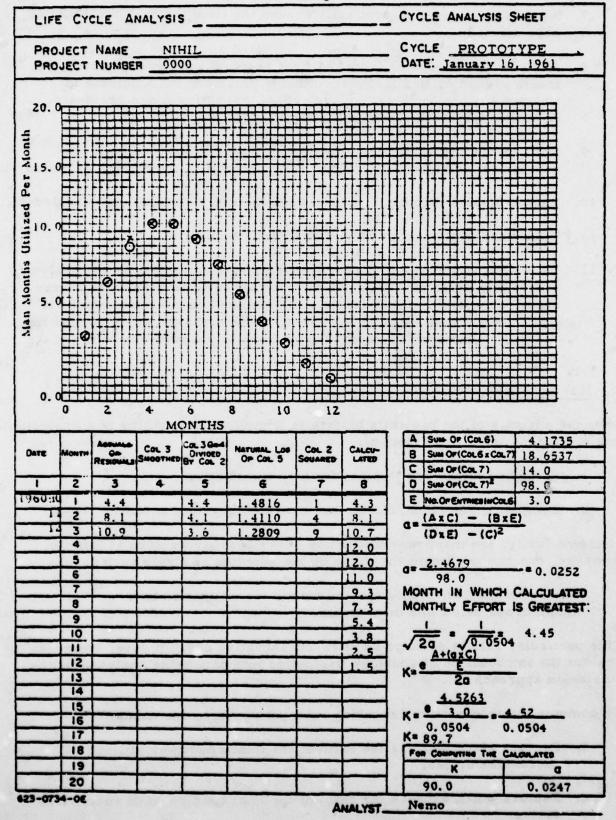


EXAMPLE 3e.



### EXAMPLE 3f





- 7. Again search for a discontinuity around the time the next cycle should have started.
- 8. Analyze the residuals, from the first residual to the residual for the month preceding this last discontinuity (Example 3f). Indicate in Column 3 of the log least squares sheet that the data are residuals. Compute this cycle also in its entirety.
- 9. Subtract the cycle just calculated from the previous residuals for another set of residuals (Example 3b: Column 5 Column 6 = Column 7).
- 10. Repeat Steps 6 through 9 if more than one cycle remains to be analyzed.
- 11. If only one cycle remains to be analyzed, analyze it.
- 12. Erroneous assumptions concerning starting month, etc. in the analysis of an early cycle can affect all later cycles. If the residuals become unworkable (i.e., if they lose all resemblance to the form which the Life-Cycle model leads us to expect), reconsider the early part of the analysis. An error in interpretation here is probably the cause of the difficulty.

## D. Making Projections

Once the analysis of the project's history is complete, the making of a projection is a simple matter. It consists of:

- 1. Extending the existing cycles to their termination.
- 2. Computing the values for the cycles which have yet to start.

Mathematically, the manpower utilization curve approaches zero at infinity. However, one can terminate the curve by the adoption of a rule such as this:

"The cycle ends when the expected manpower utilization in a month drops below one man-month."

This particular rule is perhaps an ultra-conservative one. It does, however, prevent the introduction of spurious fluctuation into data being analyzed by the outside-in approach.

To compute the values of cycles which have yet to start, one needs:

- 1. Factors which express the ratio of the total manpower utilized in successive cycles.
- 2. Factors which express the ratio of the total time spent in successive cycles.

3. A rule which positions the start of a cycle on the time stream.

To illustrate: The Design Cycle of a project required 100 man-months, spread over 15 months. The Prototype Cycle required 120 man-months, spread over a 12-month period. The factors which one obtains from this are:

K (Design Cycle) : K (Prototype Cycle) = 100: 120

or K (Design Cycle) : K (Prototype Cycle) = 1.0:1.2

and T (Design Cycle): T (Prototype Cycle) = 15:12

T (Design Cycle): T (Prototype Cycle) = 1.00:0.75

Therefore, in projecting a Prototype Cycle from a Design Cycle, one would multiply the total effort in the Design Cycle (K) by 1.2 and the total time (T) by 0.75 to find the total effort and total time in the Prototype Cycle. Pick the proper manpower utilization curve from Appendix I by inspection.

To obtain factors when first using Life-Cycle analysis, one should analyze the historical data of some fully completed projects. Exhibit I offers some average relationships which one might expect to find.

### EXHIBIT I

### Average Relationships Between Successive Cycles

Cycles	Total Manpower (K)	Total Time (T)	
Planning: Design	4.0	1.4*	
Design: Prototype	1.0	1.0	
Prototype: Release	1.0	1.0	
Release: Product Support Number One	0.4	0.7	

<sup>\*</sup> This factor seems to vary quite widely. Use of the average is only recommended if the Planning Cycle peaks in the third month or later; otherwise use best available estimates.

At times, one finds that substantial changes in the item under development cause a secondary cycle (illustrated in Figure 4). When computing the factors of this project, the total manpower for this stage is based on the total manpower in both primary and secondary cycles. The total time is computed from the start of the primary cycle to the end of the primary or secondary cycle, whichever ends latest.

About the only difficulty one encounters in making a projection is the problem of when to start a cycle. In theory as well as in practice, the project manager can exert more influence on the time he starts a cycle than on its form. However, it seems reasonable that he will try to start a cycle at a time when he can utilize the manpower he is phasing out of the previous cycle. It is suggested, therefore, that the analyst start a cycle at a time when it can absorb the manpower which is phasing out of the previous cycle, and still minimize the total manpower utilized in any given month.

## V. PREPARING A LIFE-CYCLE PROJECTION BEFORE A PROJECT STARTS

To lay out a Life-Cycle projection before a project starts, one needs three pieces of information:

- 1. For any one cycle, the months required to reach peak manpower in this cycle.
- 2. For the samecycle, the amount of manpower to be utilized at peak of cycle.
- 3. Factors which give relationships between successive cycles.

While it is possible to lay out this type of projection using any cycle as a base, the project manager can estimate  $y'_{max}$  and  $t_{y'_{max}}$  for the first cycle, the Planning Cycle, most readily because it is in the immediate future.

Using the relationships explained in Section III-A, the coefficients K and a for this base cycle are readily computed:

$$K = 1.65 (y'_{max}) (t_{y'_{max}})$$

$$a = \frac{1}{2 (t_{y'_{max}})^2}$$

T, the total time in the cycle, is approximately three times t y max. It can be determined precisely by computing the curve.

The coefficients of all other cycles can be computed by using the factors described in Section IV-D. Positioning the cycles on the calendar is done in the same manner as described in Section IV-D.

It is recognized that this projection is quite sensitive to the accuracy of the project manager's estimate of y'max and ty'max for the base cycle. However, he should be able to concentrate on this two-point estimate more easily than on a total program estimate.

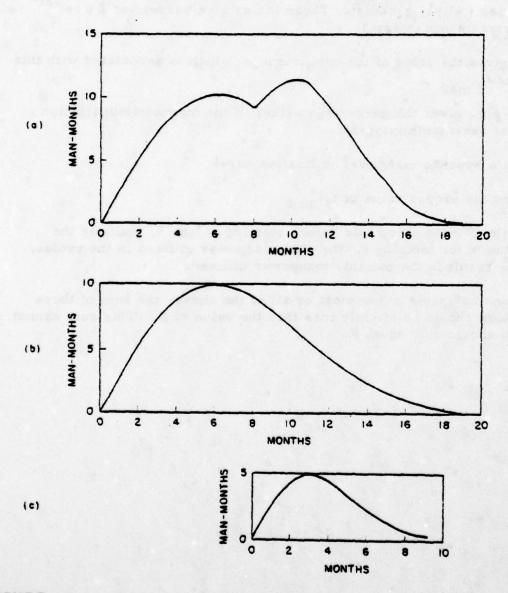


FIGURE 4. An example of a compound cycle. (a) Compound cycle as it might appear in accounting records. (b) First cycle of the compound cycle shown above. (c) Second cycle of the compound cycle shown above; note the difference in the time scale.

#### Appendix I

#### TABLES OF THE MANPOWER UTILIZATION CURVE

#### In these tables:

line 1 is the value of ty'max, the point in time at which manpower utilization (y') is greatest. These tables give curves for 2 ate at quarter-month intervals.

 $\frac{\text{line 2}}{\text{value of t}}$  gives the value of the coefficient a which is associated with this value of  $t_{y'max}$ .

line 3, etc. gives the percentage values of the manpower utilization curve at one-month intervals.

To compute a specific manpower utilization curve:

- 1. Find the proper value of ty'max
- 2. In the column below this value, starting at line 3, multiply the value in the table by K, the total manpower utilized in the cycles. The result is the monthly manpower utilized.
- 3. If one computes y' for most or all of the curve, the sum of these values should be slightly less than the value of K. This sum cannot, and should not, equal K.

No.					
MONTH OF MAXIMUM MANPOWER		1.00	1.25	1.50	1.75
THE VALUE OF A IS		0.5000	0.3200	0.2222	0.1633
(XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX	т хх	VALUE OF	THE MANPOWER	UTILIZAT	ION CURVE
	1.0 2.0 3.0 5.0 6.0 7.0	0.6065 0.2707 0.0333 0.0013 0.0000	0.4647 0.3559 0.1078 0.0153 0.0011 0.0000	0.3559 0.3654 0.1394 0.0593 0.0986 0.0909	0.2773 0.3399 0.2254 0.0958 0.0276 0.0055 0.0008

MONTH OF MAXIMUM MANPOWER	2	2.00	2.25	2.50	2.75
THE VALUE OF A IS		0.1250	0.0988	0.0900	0.0661
*********	K T XX	VALUE OF	THE MANPOWER	UTILIZAT	ION CURVE
	1.0	0.2206	0.1790	0.1477	0.1238
	2.0	0.3033	0.2661	0.2324	0.2030
	3.0	0.2435	0.2436	0.2336	0.2188
	4.0	0.1353	0.1627	0.1779	0.1836
	5.0	0.0549	0.0636	0.1093	0,1266
	6.0	0.0167	0.0339	0.0539	0.0734
	7.0	0.0038	0.0109	0.0222	0.0363
	8.0	0.0007	0.0028	0.0075	0.0154
	9.0	0.0001	0.0006	0.0022	0.0056
•	10.0	0.0000	0.0001	0.0005	0.0018
	11.0		0.0000	0.0001	0.0005
	12.0			0.0000	0.0001
	13.0				0.0000

COMPUTER SCIENCES CORP ARLINGTON VA

SOFTWARE PHENOMENOLOGY - WORKING PAPERS OF THE SOFTWARE LIFE CY-ETC(U)

AUG 77 B ELKINS, L HUNT

DAHC26-76-D-1006 AD-A053 014 UNCLASSIFIED NL 4 OF 8 ADA 053 014 製造

MONTH OF MAXIMUM MANPOWER	3.00	3.25	3.50	3.75
HE VALUE OF A IS	0.0556	0.0473	0.0408	0.0356
	XX VALUE OF	THE MANPOWER	UTILIZA	TION CURVE
1.		0.0903 0.1567	C.0794 0.1397	0.0686
3. 4.	0 0.2022	0.1855 0.1776	0.1696	0.1549
5.	0.0902	0.1450	0.1471	0.1462
7. 8. 9.	0.0254	0.0652 · 0.0366 0.0184	0.0773 0.0479 0.0259	0.0872 0.0584 0.0359
10.	0.0043	0.0083	0.0139	0.0203
12.	0.0001	0.0012	0.0027	0.0051
14.	0	0.0001	0.0004	0.0009
16.			0.0000	0.0001

MONTH OF MAXIMUM MANPOWER		4.00	4.25	4.50	4.75
THE VALUE OF A IS		0.0312	0.0277	0.0247	0.0222
*******	T XX	VALUE OF	THE MANPOWER	UTILIZAT	ION CURVE
	1.0	0.0606	0.0539	0.0492	0.0433
	2.0	0.1103	0.0991	0.0995	0.0811
	3.0	0.1415	0.1295	0.1196	0.1089
	4.0	0.1516	0.1422	0.1331	0.1244
	5.0	0.1431	0.1386	0.1332	0.1273
	6.0	0.1217	0.1226	0.1219	0.1198
	7.0	0.0946	0.0998	0.1931	0.1047
	8.0	0.0677	0.0753	0.0914	0.0859
	9.0	0.0448	0.0529	0.0601	0.0663
	10.0	0.0275	0.0348	0.0418	0.0483
	11.0	0.0157	0.0214	0.0274	0.0334
	12.0	0.0083	0.0123	0.0169	0.0219
	13.C	0.0041	0.0067	0.0099	0.0136
	14.0	0.0019	0.0034	0.0055	0.0081
	15.0	0.0008	0.0016	0.0029	0.0045
	16.0	0.0003	0.0007	0.0014	0.0024
	17.0	0.0001	0.0003	0.0007	0.0012
	18.0	0.0000	0.0001	0.0003	0.0006
	19.0		0.0000	0.0001	0.0003
	20.0			0.0001	0.0001
	21.0				0.0001

MANDAMEN MONIKAM TO HTHE		5.00	5.25	5.50	5.75
THE VALUE OF A IS		0.0200	0.0181	0.0165	0.0151
**********	T XX	VALUE OF	THE MANPOWER	UTILIZA	TION CURVE
The state of the s	1.0	0.0392	0.0356	0.0325	0.0298
D	2.0	0.0738	0.0675	0.0619	0.0569
	3.0	0.1002	0.0924	0.0355	0.0792
	4.0	0.1162	0.1086	0.1015	0.0950
U care a constant	5.0	0.1213	0.1153	0.1093	0.1036
	6.0	0.1168	0.1133	0.1994	0.1053
II	7.0	0.1051	0.1044	0.1030	0.1009
U - comment of the comment	6.C	0.0890	0.0909	0.0919	0.0919
	9.0	0.0712	0.0751	0.0789	0.0800
TI THE STATE OF TH	10.0	0.0541	0.0591	0.0633	0.0667
	11.0	0.0391	0.0444	0.0492	0.0534
	12.0	0.0269	0.0319	0.0367	0.0411
	14.0	0.0177	0.0220 0.0145	0.0191	0.0303
	15.0	0.0067	0.0092	0.0120	0.0151
	16.0	0.0038	0.0056	0.0077	0.0101
	17.0	0.0021	0.0033	0.0047	0.0065
	18.0	0.0011	0.0018	0.0029	0.0041
U STATE OF THE STA	19.0	0.0006	0.0010	0.0016	0.0024
	20.0	0.0003	0.0005	0.0009	0.0014
	21.0	0.0001	0.0003	0.0005	0.0008
U	22.0	0.0001	0.0001	0.0002	0.0004
	23.0		0.0001	0.0001	0.0002
	24.0			0.0001	0.0001
U seed of the seed of	25.0				0.0001

MONTH OF MAXIMUM MANPOWER 6.00 6.25 6.50

6.75

THE VALUE OF A IS		0.0139	0.0128	0.0119	0.0110
************************	xx	VALUE OF	THE MANPOWER	UTILIZATI	ON CURVE
8.50	. 0	0.0274	0.0253	0.0234	0.0217
2	.0	0.0526	0.0486	0.0451	0.0420
	.0	0.0735	0.0684	0.0638	0.0597
	.0	0.0890	0.0834	0.0793	0.0737
	.0	0.0981	0.0929	0.0990	0.0834
6	.0	0.1011	0.0969	0.0927	0.0887
7	.0	0.0985	0.0957	0.0929	0.0897
8	.0	0.0914	0.0903	0.0999	0.0870
9	.0	0.0812	0.0817	0.0917	0.0812
10	.0	0.0693	0.0712	0.0725	0.0732
	.0	0.0569	0.0598	0.0622	0.0640
12	. 0	0.0451	0.0486	0.0517	0.0542
13	. C	0.0345	0.0383	0.0416	0.0447
14	.0	0.0256	0.0292	0.0326	0.0358
15	. 0	0.0183	0.0216	0.0248	0.0279
16	.0	0.0127	0.0155	0.0193	0.0212
	.0	0.0065	0.0108	0.0132	0.0156
	.0	0.0056	0.0073	0.0092	0.0113
	. 0	0.0035	0.0048	0.0063	0.0079
	.0	0.0021	0.0031	0.0042	0.0054
	.0	0.0013	0.0019	0.0027	0.0036
	.0	0.0007	0.0011	0.0017	0.0024
	.0	0.0004	0.0007	0.0010	0.0015
	.0	0.0002	0.0004	0.0006	0.0009
	. 0	0.0001	0.0002	0.0004	0.0006
	.0	0.0001	0.0001	0.0002	0.0003
	.0		0.0001	0.0001	0.0002
	.0			0.0001	0.0001
29	.0				0.0001

. [						
k-i	NONTH OF MAXIMUM MANPOWER	?	7.00	7.25	7.50	7.75
	THE VALUE OF A IS		0.0102	0.0095	0.0099	0.0083
	************	K T XX	VALUE OF	THE MANPOWER	UTILIZAT	ION CURVE
		1.0	0.0202	0.0188	0.0176	0.0165
11			0.0392	0.0366	0.0343	0.0463
*		3.0	0.0559	0.0524	0.0492	0.0583
0		4.0 5.0	0.0693	0.0654	0.0617	0.0676
0			0.0791	0.0750	0.0712	0.0740
No.		6.0 7.0	0.0848	0.0810	0.0775	0.0775
-			0.0866	0.0836	0.0905	1 0.0782
		9.0	0.0804	0.0828 0.0792	0.0779	0.0763
		10.0	0.0736	0.0735	0.0731	0.0724
		11.0	0.0653	0.0662	0.0557	0.0669
n		12.0	0.0563	0.0580	0.0593	0.0603
Ц		13.0	0.0563	0.0496	0.0515	0.0530
		14.0	0.0387	0.0413	0.0435	0.0456
0		15.0	0.0308	0.0336	0.0351	0.0384
		16.0	0.0240	0.0336	0.0292	0.0316
		17.0	0.0182	0.0207	0.0232	0.0255
		18.0	0.0135	0.0157	0.0190	0.0202
		19.0	0.0097	0.0117	0.0136	0.0157
V		20.5	0.0069	0.0085	0.0192	0.0119
10		21.0	0.0048	0.0060	0.0074	0.0089
11		22-0	0.0032	0.0042	0.0053	0.0065
U		23.0	0.0021	0.0029	0-0937	0.0047
		24.0	0.0014	0.0019	0.0025	0.0033
-		25.0	0.0009	0.0012	0.0017	0.0023
Ш		26.0	0.0005	0.0008	0.0011	0.0016
		27.0	0.0003	0.0005	0.0007	0-0010
П		28.0	0.0002	0.0003	0.0005	0.0007
		29.0	0.0001	0.0002	0.0003	0.0004
		30.0	0.0001	0.0001	0.0002	0.0003
17		31.0	0.0001	0.0001	0.0001	0.0002
1		32.0		0.000	0.0001	0.0001
2.3		33.C				0.0001
						3.000.

MONTH OF MAXIMUM MANPOWER		8.00	8.25	8.50	8.75
THE VALUE OF A IS		0.0078	0.0073	0.0069	0.0065
*********	T xx	VALUE OF	THE MANPOWER	UTILIZATI	ON CURVE
	X 000000000000000000000000000000000000	VALUE OF  0.0155 0.0303 0.0437 0.0552 0.0643 0.0708 0.0746 0.0758 0.0747 0.0715 0.0668 0.0609 0.0542 0.0473 0.0404 0.0338 0.0278 0.0224 0.0177 0.0137 0.0105 0.0078 0.0058 0.0042 0.0030 0.0014 0.0010 0.0004 0.0003 0.0002 0.0001	THE MANPOWER  0.0146 0.0285 0.0413 0.0523 0.0611 0.0677 0.0718 0.0735 0.0729 0.0705 0.0664 0.0612 0.0552 0.0487 0.0422 0.0358 0.0299 0.0245 0.0197 0.0156 0.0121 0.0092 0.0069 0.0051 0.0027 0.0019 0.0013 0.0009 0.0004 0.0003 0.0002 0.0001	UTILIZATI 0.0137 0.0269 0.0399 0.0496 0.0582 0.0647 0.0699 0.0711 0.0699 0.0613 0.0659 0.0613 0.0659 0.0613 0.0659 0.0613 0.0659 0.0613 0.0659 0.0613 0.0659 0.0613 0.0659 0.0613 0.0659 0.0613 0.0659 0.0613 0.0659 0.0613 0.0659 0.0613 0.0659 0.0613 0.0659 0.0613 0.0659 0.0613 0.0659 0.0613 0.0659 0.0613 0.0659 0.0613 0.0659 0.0613	ON CURVE  0.0130 0.0254 0.0369 0.0471 0.0555 0.0619 0.0664 0.0688 0.0693 0.0680 0.0652 0.0612 0.0563 0.0451 0.0393 0.0283 0.0095
	35.0 36.0 37.0	0.0001	0.0001	0.0001	0.0002 0.0001 0.0001

L				
MONTH OF MAXIMUM MANPOWER	9.00	9.25	9.50	9.75
HE VALUE OF A IS	0.0062	0.0056	0.0055	0.0053
CXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX	VALUE OF	THE MANPOWER	UTILIZA	TION CURVE
1.0	0.0123	0.0116	0.0110	0.0105
2.0	0.0241	0.0228	0.0217	0.0206
3.0	0.0350	0.0333	0.0316	0.0301
4.0	0.0447	0.0426	0.0495	0.0387
5.0	0.0529	0.0505	0.0432	0.0461
6.0	0.0593	0.0568	0.0545	0.0522
7.0	0.0639	0.0614	0.0591	0.0569
8.0	0.0665	0.0643	0.0622	0.0601
9.0	0.0674	0.0655	0.0637	0.0618
10.0		0.0652	0.0537	0.0622
	0.0666			0.0612
11.5	0.0643	0.0634	0.0523	
12.0	0.0609	0.0605	0.0599	0.0592
13.0	0.0565	0.0566	0.0565	0.0562
14.0	0.0515	0.0520	0.0524	0.0525
15.0	0.0462	0.0471	0.0478	0.0483
10.5	0.0407	0.0419	0.0429	0.0438
17.0	0.0353	0.0367	0.0390	0.0391
18-0	0.0301	0.0317	0.0331	0.0344
19.0	0.0253	0.0269	0.0295	0.0299
20.9	0.0209	0.0226	0.0242	0.0257
21.0	0-0170	0.0187	0.0202	0.0217
22.0	0.0137	0.0152	0.0167	0.0181
23.0	0.0108	0.0122	0.0136	0.0150
24.0	0.0085	0.0097	0.0109	0.0122
25.0	0.0065	0.0076	0.0097	0.0098
26.0	0.0049	0.0058	0.0968	0.0078
27.0	0.0037	0.0045	0.0053	0.0061
28.0	0.0027	0.0034	0.0940	0.0048
29.C	0.0020	0.0025	0.0030	0.0037
30.0	0.0014	0.0018	0.0023	0.0028
31.0	0.0010	0.0013	0.0017	0.0021
32.0	0.0007	0.0009	0.0012	0.0015
33.0	0.0005	0.0007	0.0009	0.0011
34.0	0.0003	0.0005	0.0006	0.0008
35.0	0.0002	0.0003	0.0004	0.0006
36.0	0.0001			
37.0	0.0001	0.0002	0.0003	0.0004
38.0		0.0001	0.0002	0.0003
39.0	0.0001	0.0001	0.0001	0.0002
		0.0001	0.0001	0.0001
40.0			0.0001	0.0001
41.9				0.0001

MONTH OF MAXIMUM MANPOWER 10.00

10.25 10.59

10.75

THE	VALUE	OF 8 15		0.0050	0.0048	0.0045	0.0043
			<b>.</b>				
					THE MONDONED		TON CURUE
XXX	XXXXXX	XXXXXXX	XX T XXXXXXX	VALUE OF	THE MANPOWER	UTILIZAT	TON CORVE
			1.0	0.0100	0.0095	0.0090	0.0086
			2.0	0.0196	0.0187	0.0179	0.0170
			3.0	0.0287	0.0274	0.0261	0.0250
			4.0	0.0369	0.0353	0.0337	0.0323
			5.0	0.0441	0.0423	0.0495	0.0388
			6.0	0.0501	0.0481	0.0452	0.0444
			7.0		The state of the s	0.0508	0.0490
			the state of the s	0.0548	0.0528		
			8.0	0.0581	0.0562	0.0543	0.0525
			9.0	0.0600	0.0583	0.0565	0.0549
			10.0	0.0607	0.0591	0.0576	0.0561
			11.0	0.0601	0.0589	0.0576	0.0564
			12.0	0.0584	0.0576	0.0566	0.0557
			13.0	0.0558	0.0554	0.0548	0.0541
			14.0	0.0525	0.0524	0.0522	0.0519
			15.0	0.0427	0.0489	0.0490	0.0490
			16.0	0.0445	0.0450	0.0454	0.0457
			17.0	0.0401	0.0409	0.0416	0.0421
			18.0	0.0356	0.0367	0.0376	0.0383
			19.0	0.0313	0.0324	0.0335	0.0345
			20.0	0.0271	0.0284	0.0296	0.0307
			21.0	0.0232	0.0245	0.0258	0.0270
			22.0	0.0196	0.0209	0.0222	0.0235
			23.0	0.0163	0.0177	0.0199	0.0202
			24.0	0.0135	0.0147	0.0169	0.0172
			25.0	0.0110	0.0122	0.0133	0.0145
			26.0	0.0089	0.0099	0.0119	0.0121
			27.0	0.0071	0.0080	0.0099	0.0100
			28.0	0.0056	0.0064	0.0073	0.0082
	, -		29.0	0.0043	0.0050	0.0058	0.0066
			30.0	0.0033	0.0039	0.0046	0.0053
			31.0	0.0025	0.0030	0.0036	0.0042
			32.0	0.0019	0.0023	0.0029	0.0033
			33.0	0.0014	0.0018	0.0021	0.0026
			34.0	0.0011	0.0013	0.0016	0.0020
			35.0	0.0008	0.0010	0.0012	0.0015
			36.0	0.0006	0.0007	0.0009	0.0011
			37.0	0.0004	0.0005	0.0007	0.0009
			38.0	0.0003	0.0004	0.0005	0.0006
			39.0	0.0002	0.0003	0.0004	0.0005
			40.0	0.0001	0.0002	0.0003	0.0003
			41.0	0.0001	0.0001	0.0002	0.0002
			42.0	0.0001	0.0001	0.0001	0.0002
			43.0	0.0001	0.0001	0.0001	0.0001
			44.0			0.	0.0001
			45.0				0.

MONTH OF MAXIMUM MANPOWER		11.00	11.25	11.50	11.75
THE VALUE OF A IS		0.0041	0.0040	0.0038	0.0036
**********	T XX	VALUE OF	THE MANPOWER	UTILIZATIO	ON CURVE
-	1.0	0.0082	0.0079	0.0075	0.0072
	2.0	0.0163	0.0156	0.0149	0.0143
L.	3.0	0.0239	0.0229	0.0219	0.0210
	4.0	0.0309	0.0297	0.0285	0.0273
	5.0	0.0373	0.0358	0.0344	0.0381
U	7.0	0.0472	0.3456	0.0440	0.0425
	8.0	0.0508	0. 91	0.0475	0.0460
	9.0	0.0532	0 16	0.0501	0.0486
	10.0	0.0547	0.0532	0.0518	0.0504
	11.0	0.0551	0.0539	0.0526	0.0514
	12.0	0.0547	0.0537	0.0526	0.0516
	13.0	0.0534	0.0527	0.0519	0.0511
	14.0	0.0515	0.0510	0.0505	0.0499
	15.0	0.0489	0.0487	0.0494	0.0481
	16.0	0.0459	0.0460	0.0460	0.0459
	17.0	0.0426		0.0431	0.0432
	18.0	0.0390	0.0395	0.0400	0.0403
	19.0	0.0353	0.0361	0.0367	0.0372
	20.0	0.0317	0.0325	0.0333	0.0340
	21.0	0.0281	0.0291	0.0300	0.0308
	22.0 23.0	0.0246	0.0225	0.0235	0.0245
	24.0	0.0184	0.0195	0.0296	0.0216
	25.0	0.0156	0.0167	0.0173	0.0188
	26.0	0.0132	0.0142	0.0153	0.0163
	27.0	0.0110	0.0120	0.0130	0.0140
	28.0	0.0091	0.0100	0.0109	0.0119
	29.0	0.0074	0.0083	0.0091	0.0100
	30.0	0.0060	0.0068	0.0076	0.0083
	31.0	0.0048	0.0055	0.0062	0.0069
	32.0	0.0038	0.0044	0.0050	0.0057
	33.0	0.0030	0.0035	0.0041	0.0046
П	34.0 35.0	0.0024	0.0028	0.0033	0.0037
	36.0	0.0018	0.0022 0.0017	0.0026	0.0030
	37.0	0.0011	0.0013	0.0016	0.0019
	38.0	0.0008	8,0010	0.0012	0.0015
	39.0	0.0006	0.0006	0.0009	0.0011
	40.0	0.0004	0.0006	0.0007	0.0009
	41.0	0.0003	0.0004	0.0005	0.0007
	42.0	0.0002	0.0003	0.0004	0.0005
	43.0	0.0002	0.0002	0.0003 .	0.0004
	44.0	0.0001	0.0002	0.0002	0.0003
	45.0	0.0001	0.0001	0.0002	4.0002
	46.0		0.0001	0.0001	0.0002
	47. C			0.0001	0.0001
	48.C				0.0001
					,

MONTH OF MAXIMUM MANPOWER

12.00 12.25

12.50

12.75

THE VALUE OF A IS	0.0035	0.0033	0.0032	0.0031
<b>*****</b> *******************************	VALUE OF	THE MANPOWER	UTILIZAT	TION CURVE
1.0	0.0069	0.0066	0.0064	0.0061
2.0	0.0137	0.0132	0.0126	0.0122
3.0	0.0202	0.0194	0.0187	0.0180
4.0	0.0263	0.0253	0.0243	0.0234
5.0	0.0318	0.0307	0.0295	0.0285
6.0	0.0368	0.0355	0.0342	0.0330
7.0	0.0410	0.0396	0.0383	0.0370
8.0	0.0445	0.0431	0.0417	0.0404
9.0	0.0472	0.0458	0.0444	0.0432
10.0	0.0491	0.0478	0.0465	0.0452
11.0	0.0502	0.0490	0.0478	0.0466
12.0	0.0505	0.0495	0.0484	0.0474
13.0	0.0502	0.0493	0.0484	0.0476
14.0	0.0492	0.0486	0.0479	0.0471
15.0	0.0477	0.0472	0.0467	0.0462
16.0	0.0457	0.0454	0.0451	0.0448
17.0	0.0433	0.0432	0.0432	0.0430
18.0	0.0406	0.0408	0.0408	0.0409
19.0	0.0377	0.0380	0.0383	0.0385
20.0	0.0346	0.0352	0.0356	0.0359
21.0	0.0315	0.0322	0.0329	0.0333
22.0	0.0285	0.0292	0.0299	0.0305
23.0	0.0254	0.0263	0.6271	0.0278
24.0	0.0226	0.0235	0.0243	0.0251
25.0	0.0198	0.0208	0.0217	0.0225
26.0	0.0173	0.0182	0.0191	0.0200
27.0	0.0149	0.0159	0.0168	0.0176
28.0	0.0128	0.0137	0.0145	0.0154
29.0	0.0109	0.0117	0.0126	0.0134
30.0	0.0092	0.0100	0.0108	0.0116
31.0 32.0	0.0077	0.0084	0.0092	0.0099
33.0	0.0063	0.007 <b>0</b> 0.0058	0.0077	0.0084
34.0	0.0043	0.0038	0.0065	0.0071
35.0	0.0035	0.0039	0.0034	0.0050
36.0	0.0028	0.0039	0.0036	0.0041
37.0	0.0022	0.0026	0.0030	0.0034
38.0	0.0018	0.0021	0.0024	0.0028
39.0	0.0014	0.0016	0.0019	0.0022
40.0	0.0011	0.0013	0.0015	0.0018
41.0	0.0008	0.0010	0.0012	0.0014
42.0	0.0006	0.0008	0.0010	0.0011
43.0	0.0005	0.0006	0.0007	0.0009
44.0	0.0004	0.0005	0.0006	0.0007
45.0	0.0003	0.0004	0.0004	0.0005
46.0		0.0003	0.0003	0.0004
47.0		A FLIDT	0.0003	0.0003
46.0				0.0002

\$10.4					
MONTH OF MAXIMUM MANPOWER		13.00	13.25	13.50	13.75
HE VALUE OF A IS		0.0030	0.0028	0.0027	0.0026
r					
_ <b>(XXXXXXXXXXXXXXXXX</b> XXXXXXX	T XX	VALUE OF	THE MANPOWER	UTILIZA	TION CUPVE
n	1.0	0.0059	0.0057	0.0055	0.0053
	2.0	0.0117	0.0113	0.0109	0.0105
	3.0	0.0173	0.0167	0.0161	0.0155
	4.0	0.0226	0.0218	0.0210	0.0203
	5.0	0.0275		0.0256	0.0248
	6.0	0.0319	0.0308	0.0298	0.0289
П	7.0	0.0358	0.0347	0.0336	0.0325
	8.0	0.0392	0.0380	0.0368	0.0357
	9.0	0.0419	0.0407 0.0428	0.0395	0.0384 0.0406
	0.0	0.0440	0.0444	0.0433	0.0422
	2.0	0.0464	0.0454	0.0444	0.0434
	3.0	0.0467	0.0458	0.0449	0.0440
	4.0	0.0464	0.0456	0.0449	0.0441
	5.0	0.0456	0.0450	0.0444	0.0438
	6.0	0.0444	0.0440	0.0435	0.0430
	7.0	0.0428	0.0425	0.0422	0.0419
	8.0	0.0408	0.0407	0.0405	0.0404
	9.0	0.0386	0.0387 0.0365	0.0387	0.0387 0.0367
7	1.0	0.0382	0.0361	0.0344	0.0346
	2.0	0.0311	0.0316	0.0320	0.0324
	3.0	0.0235	0.0290	0.0296	0.0300
	4.0	0.0258		0.0271	0.0277
2 2	5.0	0.0233	0.0240	0.0247	0.0253
	6.0	0.0208		0.0223	0.0230
7	7.0	0.0185	0.0193	0.0200	0.0208
	9.0	0.0163	0.0171	0.0179	0.0186
	0.0	0.0143	0.0151 0.0132	0.0158	0.0166
	1.0	0.0107	0.0114	0.0122	0.0129
	2.0	0.0092	0.0099	0.0106	0.0113
	3.0	0.0078	0.0085	0.0091	0.0098
	4.0	0.0066	0.0072	0.0073	0.0085
	5.0	0.0055	0.0061	0.0067	0.0073
	6.0	0.0046	0.0051	0.0056	0.0062
	7.0	0.0038	0.0043	0.0047	0.0052
	9.0	0.0031	0.0035	0.0040	0.0044
	0.0	0.0021	0.0024	0.0027	0.0037
	1.0	0.0017	0.0019	0.0022	0.0025
	2.0	0.0013	0.0016	0.0018	0.0021
A 100 M	3.0	0.0011	0.0013	0.0015	0.0017
	4.0	0.0008	0.0010	0.0012	0.0014
	5.0	0.0007	0.0008	0.0010	0.0011
	6.0		0.0006	0.0008	0.0009
	7.0			0.0006	0.0007
	3.0	297			0.0006

MONTH OF MAXIMUM MANPOWER	14.00	14.25	14.50	14.75
THE VALUE OF A IS	0.0026	0.0025	0.0024	0.0023
**********************	VALUE OF	THE MANPOWER	UTILIZAT	TION CURVE
1.9	0.0051	0.0049	0.0047	0.0046
2.0	0.0101	0.0098	0.0094	0.0091
3.0	0.0150	0.0144	0.0140	0.0135
4.0	0.0196	0.0189	0.0193	0.0177
5.0	0.0239	0.0232 0.0270	0.0224	0.0217
7.0	0.0215	0.0306	0.0296	0.0287
8.0	0.0347	0.0337	0.0327	0.0317
9.0	0.0373	0.0363	0.0353	0.0343
10.0	0.0395	0.0385	0.0375	0.0365
11.0	0.0412	0.0402	0.0392	0.0383
12.0	0.0424	0.0415	0.0405	0.0396
13.0	0.0431	0.0422	0.0414	0.0405
14.0	0.0433	0.0426	0.0418	0.0410
15.0	0.0431	0.0424	0.0418	0.0411
16.0	0.0425	0.0420	0.0414	0.0408
17.0 18.0	0.0415	0.0411	0.0407	0.0402
19.0	0.0402 0.0386	0.0399 0.038 <b>5</b>	0.0396	0.0393
20.0	0.0368	0.0368	0.0367	0.0367
21.0	0.0348	0.0349	0.0350	0.0350
22.0	0.0327	0.0329	0.0331	0.0332
23.0	0.0304	0.0308	0.0311	0.0313
24.0	0.0282	0.0286	0.0299	0.0294
25.0	0.0259	0.0264	0.0269	0.0273
26.0	0.0236	0.0242	0.0249	0.0253
27.0	0.0215	0.0221	0.0227	0.0232
28.0 29.0	0.0193	0.0200	0.0206	0.0212
30.0	0.0173	0.0180 0.0161	0.0187	0.0193
31.0	0.0136	0.6143	0.0150	0.0157
32.0	0.0120	0.0127	0.0133	0.0140
33.0	0.0105	0.0111	0.0113	0.0124
34.0	0.0091	0.0097	0.0103	0.0110
35.0	0.0078	0.0084	0.0090	0.0096
36.0	0.0067	0.0073	0.0079	0.0084
37.0	0.0057	0.0063	0.0068	0.0073
38.0	0.0049	0.0053	0.0058	0.0063
39.0 40.0	0.0041	0.0045	0.0050	0.0054
41.0	0.0029	0.0038 0.0032	0.0036	0.0047 0.0040
42.0	0.0024	0.0032	0.0030	0.0033
43.0	0.0020	0.0022	0.0025	0.0028
44.0	0.0016	0.0018	0.0021	0.0024
45.0	0.0013	0.0015	0.0017	0.0020
46.0		0.0012	0.0014	0.0016
47.0			0.0012	0.0013
48.0				0.0011

П					
MONTH OF MAXIMUM MANPOWER		15.00	15.25	15.50	15.75
THE VALUE OF A IS		0.0022	0.0021	0.0021	0.0020
THE THEOR OF H 13		0.0022			
n					
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX	T XX	VALUE OF	THE MANPOWER	UTILIZATI	ON CURVE
	1.0	0.0044	0.0043	0.0042	0.0040
	2.0	0.0088	0.008 <b>5</b> 0.0127	0.0093	0.0119
	3.0 4.0	0.0131	0.0166	0.0151	0.0156
	5.0	0.0210	0.0204	0.0198	0.0192
	6.0	0.0246	0.0239	0.0232	0.0225
	7.0	0.0279	0.0271	0.0263	0.0256
	8.0	0.0308	0.0300	0.0291	0.0283
	9.0	0.0334	0.0325	0.0315	0.0308
	0.0	0.0356	0.0347	0.0339	0.0330
Π 1	1.0	0.0374	0.0365	0.0356	0.0347
	2.0	0.0387	0.0379	0.0370	0.0362
	3.C	0.0397	0.0389	0.0381	0.0373
Ti and the second secon	4.0	0.0403	0.0395	0.0399	0.0380
	5.0	0.0404	0.0398	0.0391	0.0384
	6.C 7.C	0.0403	0.0397 0.0393	0.0391	0.0385
and the second s	8-0	0.0399	0.0396	0.0392	0.0378
	9.0	0.0379	0.0376	0.0373	0.0370
	0.0	0.0365	0.0364	0.0362	0.0360
	1.0	0.0350	0.0350	0.0349	0.0348
	2.0	0.0334	0.0334	0.0334	0.0334
2	3.0	0.0316	0.0317	0.0318	0.0319
2	4.0	0.0297	0.0299	0.0301	0.0303
	5.0	0.0277	0.0280	0.0283	0.0286
	6.0	0.0257	0.0261	0.0255	0.0268
	7.0	0.0237	0.0242	0.0246	0.0250
	8.0	0.0218	0.0223	0.0229	0.0232
	9.0	0.0199	0.0204	0.0210	0.0215
	1.0	0.0163	0.0186	0.0192	0.0197
	2.0	0.0146	0.0152	0.0158	0.0164
	3.0	0.0130	0.0137	0.0142	0.0148
	4.0	0.0116	0.0122	0.0129	0.0133
	5.0	0.0102	0.0108	0.0114	0.0119
	6.0	0.0090	0.0095	0.0101	0.0106
	7.0	0.0078	0.0084	0.0069	0.0094
	8.0	0.0068	0.0073	0.0078	0.0083
	9.0	0.0059	0.0064	0.0068	0.0073
	0.0	0.0051	0.0055	0.0060	0.0064
	1.0	0.0043	0.0047	0.0052	0.0056
	2.0 3.0	0.0037	0.0041	0.0044	0.0048
	4.0	0.0036	0.0039	0.0033	0.0042
	5. C	0.0022	0.0025	0.0029	0.0031
2.4	6.0	3.0022	0.0021	0.0023	0.0026
	7.0			0.0020	0.0022
11	6.0				0.0019
Ad .		200			

#### Appendix II

# HOW TO TAKE A NATURAL LOGARITHM - WITH TABLES - AND HOW TO CONVERT A NATURAL LOGARITHM TO A NUMBER

To take the natural logarithm of a number, using the tables given in this Appendix:

- 1. If the number is between 1.00 and 9.99:
  - a. Look down the extreme lefthand column of figures until you locate the first two digits of the number.
  - b. Follow across this row until you locate the column heading corresponding to the next digit.
  - c. The figures at the intersection of the row containing the first two digits and the column containing the third digit are the natural logarithm of the number.

### To illustrate: Take the natural log of 5.87:

- a. Look down the extreme left-hand column for 5.8.
- b. Go across this row to the column with 7 at the top.
- c. The natural logarithm is 1.7699.
- 2. If the number is between 10.0 and 99.9:
  - a. Divide the number by 10.0.
  - b. Look up the natural logarithm of the result, which will be between 1.00 and 9.99. The procedure here is illustrated above.
  - c. Add 2.3026 to the result. 2.3026 is the natural logarithm of 10.0. Adding this to the logarithm taken from the table multiplies it by 10.0, thus raising this to the logarithm of the original number.

## To illustrate: Take the natural logarithm 58.76.

a. Divide 58.76 by 10.0 to get 5.876.

- b. Look up 5.87 as indicated above. The last digit (6) may be dropped as insignificant to the context of Life-Cycle analysis, or obtained by taking 6/10 of the difference between the log of 5.87 (1.7699) and the log of 5.88 (1.7716) and adding this to the log of 5.87, i.e., 0.0017 x 0.6 = 0.00102 and 1.7699 + 0.0010 = 1.7709.
- c. Add 2.3026 to this logarithm to obtain the log of the original number, i.e., 1.7709 + 2.3026 = 4.0735 which is the natural log of 58.76.
- 3. If the number is between 0.100 and 0.999:
  - a. Multiply the number by 10.0. The result will be a number between 1.00 and 9.99.
  - b. Look up the natural logarithm in the table following the procedure outlined above.
  - c. Subtract 2.3026 from the figure given in the table. The result will be negative. This is the natural logarithm of the original number.

To illustrate: Take the natural logarithm of 0.587.

- a. Multiply 0.587 by 10.0 to get 5.87.
- b. Look up the natural logarithm (1.7699) of 5.87.
- c. Subtract 2.3026 from this figure:
  - 1.7699 2.3026= -0.6327. This is the natural logarithm of 0.587.
- 4. If the number is 100.0 or greater, or 0.0999 or less, simply repeat the process described in (2) and (3) until the number falls between 1.00 and 9.99. Add or subtract 2.3026 to the tabled value the same number of times you divided or multiplied the original number by 10.0 to bring it within the limits of 1.00 and 9.99.

To illustrate: Take the natural logarithm of:

Total Control

a. 
$$587.0 = 1.7699 + 2.3026 + 2.3026 = 6.3751$$

- c. 0.0587 = 1.7699 2.3026 2.3026 = -2.8353
- d. 0.00587 = 1.7699 2.3026 2.3026 - 2.3026 =-5.1379

To convert a natural logarithm to a number, using the tables given in this Appendix:

- 1. If the natural logarithm is at least 0.0000 but less than 2.3026:
  - a. Find the logarithm in the body of the table.
  - b. The first two digits of the number are those in the extreme left-hand column which identify the row containing the logarithm. The third digit of the number is the one which identifies the column containing the logarithm.

To illustrate: Find the number whose logarithm is 1.7699.

- a. Locate the logarithm in the table.
- b. Note the number that identifies the row contains the logarithm (5.8). These are the first two digits of the number.
- c. Note the number that identifies the column containing the logarithm (7). This is the third digit of the number, which is 5.87.

Note: The logarithm you must convert to a number will generally not appear in the table. Usually sufficient accuracy can be obtained by finding the number of the tabled logarithm nearest to the one you have. Greater accuracy affects only the third digit of the number. Greater accuracy can be obtained by interpolating between tabled values.

- 2. If the logarithm is 2.3026 or greater:
  - a. Subtract 2.3026 from the logarithm as many times as is necessary to bring the logarithm within the range of the table (0.0000 to 2.3025).
  - b. Find the number, following the procedure given above.
  - c. Multiply the result by 10.0 as many times as you subtracted 2.3026.

To illustrate: Find the number whose natural logarithm is 6.3751.

- a. 6.3751 2.3026 = 4.0725
- b. 4,0725 2,3026 = 1,7699
- c. 1.7699 in the log of 5.87
- d.  $5.87 \times 10.0 \times 10.0 = 587.0$ , which is the number whose natural logarithm is 6.3751.
- 3. If the logarithm is less than 0.0000 (i.e., if it is negative):
  - a. Add 2.3026 as many times as is necessary to make the the logarithm a positive number within the limits of the table.
  - b. Find the number, following the procedure outlined above.
  - c. Divide this number by 10.0 as many times as you added 2.3026.

To illustrate: Find the number whose natural logarithm is - 2.8353.

- a. -2.8353 + 2.3026 = -0.5327
- b. -0.5327 + 2.3026 = 1.7699
- c. 1.7699 is the log of 5.87.
- d. 5.87  $\frac{.}{.}$  10.0  $\frac{.}{.}$  10.0 = 0.587, which is the number whose natural log is -2.8353.

## TABLES OF NATURAL LOGARITHMS

(See )

	0	1	2	3	4	5	6	7	8	9
1.0 0		0.0100	0.0199	0.0296	0.0392	0.0488	0.0593	9.0677	0.0770	0.0862
1.1 0	0.0953	0.1044	0.1133	0.1222	0.1310	0.1398	0.1494	0.1570	0.1655	0.1740
1.2 0	1823	0.1906	0.1989	0.2070	0.2151	0.2231	0.2311	0.2390	0.2469	9.2546
1.3 0	2624	0.2700	0.2776	0.2852	0.2927	0.3001	0.3075	0.3148	0.3221	0.3293
1.4 0	3365	0.3436	0.3507	0.3577	0.3646	0.3716	0.3794	0.3853	0.3920	0.3988
1.5 0	. 4055	0.4121	0.4197	0.4253	0.4318	0.4383	0.4447	0.4511	0.4574	9.4637
				0.4886	0.4947	0.5008	0.5068	9.5128	0.5188	0.5247
		0.5365			0.5539	0.5596	0.5653	0.5710	0.5766	0.5822
		0.5933					0.6296			
		0.6471		0.6575	0.6627	0.6678	0.6729	0.6780	0.6831	0.6881
	3.6931						0.7227			
		0.7467					0.7701			
		0.7930					0.8154			
							0.9002			
2.4 0	0.0167	0.8795	0.0035	0.0079	0.6720	0.6961	0.9400	0.9042	0.9003	0.9517
2 6 0	0555	0.9593	0.9243	0.9470	0.9322	0.9361	0.9753	0.9821	0.9410	0.9895
							1.0152			
							1.0508			1.0613
							1.0852			
							1.1194			
	1.1314	1.1346	1.1378	1.1410	1.1442	1.1474	1.1506	1.1537	1.1569	1.1600
							1.1817			1.1909
		1.1969					1.2119			
							1.2413			1.2499
							1.2698			1.2782
3.6 1	. 2809						1.2975			1.3056
3.7 1	1.3083	1.3110	1.3137	1.3164	1.3191	1.3218	1.3244	1.3271	1.3297	1.3324
3.8 1	.3350	1.3376	1.3402	1.3429	1.3455	1.3481	1.3507	1.3533	1.3558	1.3584
3.9 1	1.3610	1.3635	1.3661	1.3686	1.3712	1.3737	1.3762	1.3788	1.3813	1.3838
							1.4012			1.4085
4.1 1	1.4110			1.4183			1.4255			1.4327
4.2 1	4351						1.4493			1.4563
4.3 1	1.4586	1.4609	1.4633				1.4725			1.4793
		1.4839	The second secon				1.4951			1.5019
	1.5041		1.5085		1.5129		1.5173			1.5239
	.5261		1.5304				1.5390			1.5454
	1.5476			1.5539			1.5602			1.5665
		1.5707					1.5810			1.5672
	1.5892						1.6014		1.6054	
		1.6114	1.6134	1.6154	1,6174	1.6194	1.6214	1.6233	1.6253	1.6273
	6292		1.6332				1.6409			
-		1.6506					1.6601			
							1.6790			1.6845
3.4	0004	1.0002	1.0271	1.0717	1,0738	1.0736	1.07/4	1.0333	1. /011	1.7029

#### TABLES OF NATURAL LOGARITHMS

```
9
        0
                         2
                                 3
                                                  5
                                                           6
                                                                    7
                                                                            8
5.5 1.7047 1.7066 1.7094 1.7102 1.7120 1.7138 1.7156 1.7174 1.7192 1.7210
             1.7245 1.7263 1.7281 1.7299 1.7317 1.7334 1.7352 1.7370 1.7387
5.6 1.7228
                                       1.7475 1.7492 1.7509 1.7527 1.7544 1.7561
     1.7405 1.7422 1.7440 1.7457
             1.7596 1.7613 1.7630 1.7647 1.7664 1.7681 1.7699 1.7716 1.7733
5.8 1.7579
5.9 1.7750 1.7766 1.7783 1.7800 1.7817 1.7834 1.7851 1.7867 1.7884 1.7901
6.0 1.7918 1.7934 1.7951 1.7967 1.7984 1.8001 1.8017 1.8034 1.805u 1.9066
6.1 1.8083 1.8099 1.8116 1.8132 1.8148 1.8165 1.8191 1.8197 1.8213 1.9229
6.2 1.8245 1.8262 1.8279 1.8294 1.8310 1.8326 1.8342 1.8358 1.8374 1.9390
6.3 1.8405 1.8421 1.8437 1.8453 1.8469 1.8485 1.8500 1.8516 1.8532
                                                                                  1.3547
6.4 1.8563 1.8579 1.8594 1.8610 1.8625 1.8641 1.8656 1.8672 1.8687 1.9703
6.5 1.8718 1.8733 1.8749 1.8764 1.8779 1.8795 1.8810 1.8825 1.8840 1.9856
6.6 1.8871 1.8886 1.8901 1.8916 1.8931 1.8946 1.8951 1.8976 1.8991 1.9006
 6.7 1.9021 1.9036 1.9051 1.9066 1.9081 1.9095 1.9110 1.9125 1.9140 1.9155
 6.8 1.9169 1.9184 1.9199 1.9213 1.9228 1.9242 1.9257 1.9272 1.9286 1.9301
6.9 1.9315 1.9330 1.9344 1.9359 1.9373 1.9387 1.9402 1.9416 1.9430
                                                                                  1.9445
7.0 1.9459 1.9473 1.9488 1.9502 1.9516 1.9530 1.9544 1.9559 1.9573 1.9587
 7.1 1.9601 1.9615 1.9629 1.9643 1.9657 1.9671 1.9685 1.9699 1.9713 1.9727
7.2 1.9741 1.9755 1.9769 1.9782 1.9796 1.9810 1.9824 1.9838 1.9851 1.9865
7.3 1.9879 1.9892 1.9906 1.9920 1.9933 1.9947 1.9961 1.9974 1.9988 2.0001
 7.4 2.0015 2.0028 2.0042 2.0055 2.0069 2.0082 2.0096 2.0109 2.0122 2.0136
7.5 2.0149 2.0162 2.0176 2.0189 2.0202 2.0215 2.0229 2.0242 2.0255 2.0268 7.6 2.0281 2.0295 2.0308 2.0321 2.0334 2.0347 2.0360 2.0373 2.0386 2.0399
7.7 2.0412 2.0425 2.0439 2.0451 2.0464 2.0477 2.0499 2.0503 2.0516 2.9528 7.8 2.0541 2.0554 2.0567 2.0580 2.0592 2.0605 2.0618 2.0631 2.0643 2.0656 7.9 2.0669 2.0681 2.0694 2.0707 2.0719 2.0732 2.0744 2.0757 2.0769 2.0782 8.0 2.0794 2.0807 2.0819 2.0832 2.0844 2.0857 2.0869 2.0882 2.0894 2.0906
 8.1 2.0919 2.0931 2.0943 2.0956 2.0968 2.0980 2.0992 2.1005 2.1017 2.1029
8.2 2.1041 2.1054 2.1066 2.1078 2.1090 2.1102 2.1114 2.1126 2.1138 2.1150
 8.3 2.1163 2.1175 2.1187 2.1199 2.1211 2.1223 2.1235 2.1247 2.1258 2.1270
 8.4 2.1282 2.1294 2.1396 2.1318 2.1330 2.1342 2.1353 2.1365 2.1377 2.1389
 8.5 2.1401 2.1412 2.1424 2.1436 2.1448 2.1459 2.1471 2.1483 2.1494 2.1506
 8.6 2.1518 2.1529 2.1541 2.1552 2.1564 2.1576 2.1597 2.1599 2.1610 2.1622
 8.7 2.1633 2.1645 2.1656 2.1668 2.1679 2.1691 2.1792 2.1713 2.1725 2.1736
 8.8 2.1748 2.1759 2.1770 2.1782 2.1793 2.1804 2.1815 2.1827 2.1838 2.1849
 8.9 2.1861 2.1872 2.1883 2.1894 2.1905 2.1917 2.1928 2.1939 2.1950 2.1961
 9.0 2.1972 2.1983 2.1994 2.2006 2.2017 2.2028 2.2039 2.2050 2.2061 2.2072
 9.1 2.2083 2.2094 2.2105 2.2116 2.2127 2.2138 2.2149 2.2159 2.2170 2.2181
 9.2 2.2192 2.2203 2.2214 2.2225 2.2235 2.2246 2.2257 2.2268 2.2279 2.2289
 9.3 2.2300 2.2311 2.2322 2.2332 2.2343 2.2354 2.2364 2.2375 2.2386 2.2396
     2.2407 2.2418 2.2428 2.2439 2.2450 2.2460 2.2471 2.2481 2.2492 2.2502 2.2513 2.2523 2.2534 2.2544 2.2555 2.2565 2.2576 2.2586 2.2597 2.2607
     2.2618 2.2628 2.2638 2.2649 2.2659 2.2670 2.2680 2.2690 2.2701 2.2711
9.7 2.2721 2.2732 2.2742 2.2752 2.2762 2.2773 2.2793 2.2793 2.2803 2.2814 9.8 2.2824 2.2834 2.2844 2.2854 2.2865 2.2875 2.2895 2.2895 2.2905 2.2915 9.9 2.2925 2.2935 2.2946 2.2956 2.2966 2.2976 2.2986 2.2996 2.3006 2.3016 10.0 2.3026 2.3036 2.3046 2.3056 2.3066 2.3076 2.3096 2.3096 2.3106 2.3115
```

### REFERENCES

- 1. Eisenhut, P.S. "New Insights into the Life Cycle Approach", AIIE Transactions, Vol.5, No.2, June 1973
- 2. GUIDE International, Inc. "System Development Life Cycle, a Methodology" Produced by the Installation Guidelines and Standards Group, Languages and Standards Division (in preparation)
- 3. Norden, P.V. "Useful Tools for Project Management", in Management of Production,
  M.K. Starr, Ed. Penguin, 1970
- 4. Norden, P.V. "Applications of the Life Cycle Model to Project Manpower Planning and Control" in Manpower Planning, NATO Science Committee Conference, Brussels; The English Universities Press, Ltd., 1966
- 5. Norden, P.V. "Manpower Utilization Patterns in Research and Development Projects",

  Technical Report TR 00.1191, 9/1/64, IBM Corporation, Poughkeepsie, N.Y.
- 6. Morden, P.V. and Bakshi, A.V. "Internal Dynamics of Research and Development Projects", in Management Sciences Models and Techniques, Pergamon Press, 1960
- 7. Norden, P.V. "On the Anatomy of Development Projects", <u>IRE Transactions</u>, P.G.E.M., Volume EM - 7, No. 1, March 1960
- 8. Norden, P.V. "Curve Fitting for a Model of Applied Research and Development Scheduling", IBM Journal of Research and Development, Vol. 2, No.3, July 1958
- 9. Strassman, Paul A. "Stages of Growth", Datamation, Vol. 22, No.10, October 1976

# THE INFLUENCE OF THE TIME - DIFFICULTY FACTOR IN LARGE SCALE SOFTWARE DEVELOPMENT

LAWRENCE H. PUTNAM US Army Computer Systems Command Fort Belvoir, VA 22060

#### ABSTRACT

People concerned with large scale application software development have long known that the ability to accelerate the development of a system even by adding more manpower, is almost always unsuccessful. Quantitative measures are developed in the paper to show why this is so and that a system has a natural development time which depends upon its inherent difficulty.

#### INTRODUCTION

Development of large scale application software systems has been characterized by severe cost overruns and time slippages from a predetermined schedule. This is usually because of the implicit assumption that the development effort is the product of people and time and that the time can be set arbitrarily by management and thus the manning level is simply the development effort in manyears divided by the predetermined development time. Figure 1 below illustrates this concept.

It is clear that we are dealing with time rates of doing software work. All of the formulations above assume that the rates--productivity ( $S_{S_j}MY$ ) or manpower (People/yr)--is constant throughout the development period.

#### It will be shown that:

- 1. These relations are too simple to handle all but small programs.
- 2. Productivity rates are not constant but a complex function of the system difficulty.
- Manpower rates are not constant but also a function of the system difficulty.

It has been empirically established by examining more than 100 large system (25-1000 MY development effort, 2-5 years to develop) that systems follow a life cycle pattern in manpower and cumulative effort that look like this:

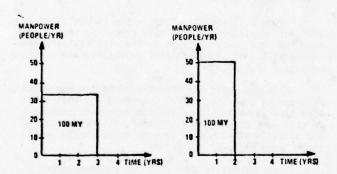


Figure 1. Two different applications of manpower to a 100 MY development effort.

Another way of treating the problem is to estimate the total number of source lines of code from the specifications then to use some overall productivity rates from previous similar projects and then divide the source code estimate by the productivity rate to get a manyear number. Time or manpower then obtained by a process similar to figure 1. For example, assume that  $S_{\rm S}$  = 100,000 lines of source code.

 $\overline{PR}$  = 1000 S<sub>S</sub>/MY by analogy with a previous project. Then development effort =  $\frac{100,000 \text{ S}_S}{1,000 \text{ S}_S/MY}$  = 100 MY

If manpower is constrained to 25 people then time would be determined as  $t_d = \frac{100 \text{ MY}}{25 \text{ people}} = 4 \text{ years, or if}$  the system is needed in two years then the relation becomes Manpower =  $\frac{100 \text{ MY}}{2 \text{ yrs}} = 50 \text{ people over the two}$  year period, or a rate of 50 people/yr.

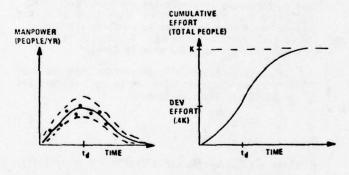


Figure 2. Manpower and cumulative effort as a function of time for software development.

The data points are shown on the manpower figure to indicate that there is scatter or "noise" involved in the process. Empirical evidence suggests that the "noise" component may be up to  $\pm$  25% of the expected manpower value during the rising part of manpower curve which corresponds to the development effort.  $t_d$  denotes the time of peak effort and is very close to the development time for the system. The falling part of the manpower curve corresponds to the operations and maintenance phase of the system life cycle. The principal work during this phase is modification, minor enhancement and remedial repair (fixing "bugs").

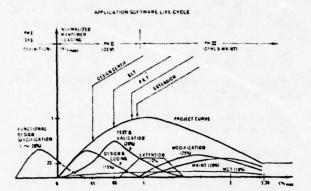


Figure 3. System Life Cycle model with component cycles and major milestones.

Note that all the subcycles (except extension) have continuously varying rates and have long tails indicating that the final 10% of each phase of effort takes a relatively long time to complete (this explains the "90% done" paradox. A subcycle is 90% done in work, but only 2/3 done in time).

It has been empirically determined that the overall life cycle manpower curve can be well represented by curves of the Rayleigh form.

$$y' = 2Kate^{-at^2}$$
 (1)

where 
$$a = 1/2 t_d^2$$
 and (2)

K is the area under the curve from t = 0 to infinity.

Making the substitution for a, we have

$$y' = K/t_d^2 \cdot t \cdot e^{-t^2/2t_d^2}$$
 (3)

and the definite integral of this is

$$y = K(1-e^{-t^2/2ta^2})$$
, which (4)

is the cumulative number of people used by the system at any time, t.

Now we wish to examine the parameters K and td:

K is the area under the y curve. It is the total number of people used by the project over its life cycle.

 $t_d$  is the time the curve reaches a maximum. Empirically this is very close to the time a system becomes operational and it will be assumed hereafter that  $t_d$  = development time for a system.

Most software cost is directly proportional to people cost (even computer testing time is proportional to the test and validation manpower).

Thus, the life cycle cost of a system is:

$$SLC = \frac{SCost/MY}{K}$$
 (5)

Neglecting cost of computer test time, inflation over time, etc., all of which can be easily handled by simple extensions of the basic ideas. The development cost is simply the  $\frac{SCost/MY}{T}$  times the area under the Y curve from 0 to  $t_d$ .

That is,

S Dev = 
$$\frac{\text{SCost/MY}}{\text{SCost/MY}}$$
 .  $\int_0^{\text{td}} y' . dt$   
=  $\frac{\text{SCost/MY}}{\text{SCost/MY}}$  . (.3945 K)

\$ Dev = 40% \$LC. (6)

These are the management parameters of the system - the people, time and dollars. Since they are related as functions of time, we then have cumulative people and cost as well as yearly (or instantaneous) people and cost at any point during the life cycle.

In the real world where requirements are never firmly fixed and changes to specifications are occurring at least to some extent, the parameters K and  $t_d$  are not completely fixed or deterministic. There is "noise" introduced into the system by the continuous human interaction. Thus, the system has random or stochastic components superimposed on the deterministic behavior. So, we are really dealing with expected values for y, y and the cost functions with some "noise" superimposed.

Figure 2 illustrates this -- the random character will not be pursued further here except to say it has important and practical uses in the risk analysis associated with initial operational capability and development cost (2,4,5).

Let's now shift our attention from the empirical evidence to some theoretical underpinnings that will shed more light on the software development process. We will look at the Rayleigh Model, its parameter and dimentions and how it appears to relate to well established physical principles.

In particular we will consider the management  $\clubsuit$  parameters K and  $t_d$  so that we may study the impact of them on the system behavior.

Figure 4 shows the software development process as a transformation. The output quantity of source statements is highly variable from project to project. This implies that there are losses in the process. These losses would also be highly variable if the energy (work) relation is obeyed.

#### SOFTWARE DEVELOPMENT PROCESS (TRANSFORMATION PROCESS)

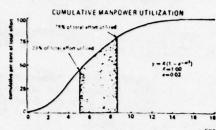


Figure 4. Software Development as a Transformation.

Newton's Law of Conservation of Energy-Work in = Work Out = Output Quantity of  $S_{\rm S}$  + Losses

Figure 5 shows the Rayleigh Model in both its integral and derivative form.





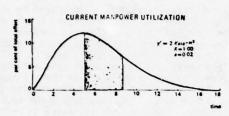


Figure 5. (1)

The derivative equation is this:

$$y' = 2Kate-at^2$$
 (7)

Where K is the total life cycle manyears of the process and a is the shape parameter in mathematical terms.

But a is more fundamental in a physical sense.

 $a=1/2\ t_d^2$ , where  $t_d$  is the time to reach peak effort. In terms of software projects  $t_d$  has been empirically shown to correspond very closely to the design time (or the time to reach initial operational capability) of a large software project. Now we can write the Rayleigh equation with the parameter  $t_d$  shown explicitly by substituting for a:

$$y' = K/t_0^2 t e^{-t^2/2t_0^2}$$
 (8)

Now it is instructive to examine the dimensions of the parameters.

K is total work done on the system.

td and t are time units.

 $\frac{K}{t_d}^2$  x t has the dimensions of power, i.e., the time rate of doing work on the system, or manpower.

The cumulative work done on the system is Power x Time or

Y = 
$$\int_{t_1}^{t_2} \int_{0}^{t} dt$$
  
Y =  $\int_{0}^{t} 2Kate^{-at^2}$   
Y = K (1-e<sup>-at^2</sup>) MY (9)

But the ratio  $K/t_d^2$  appears to have more fundamental significance. Its dimensions are those of force.

When the Rayleigh equation is linearized by taking logarithms

$$\log (y/t) = \log (K/t_d^2) + \frac{(-\log e)}{2t_d^2} t^2$$
 (10)

and then plotted as the manpower applied to a system over  ${\rm time}^2$  a straight line is produced:

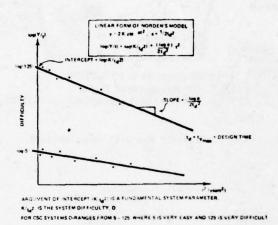
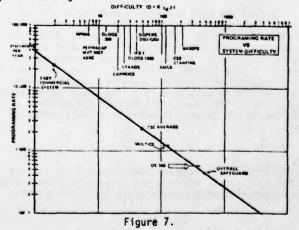


Figure 6.

When this was done for some 40 odd systems, it was found that the argument of the intercept,  $K/t_d^2$ , had a most interesting property. If the Number  $K/t_d^2$  was small it corresponded with easy systems; if the number was large, it corresponded with hard systems and appeared to follow a continuum in between. This suggested that the ratio  $K/t_d^2$  represented the difficulty of a system in terms of the programming effort to produce it.

When this ratio was plotted against the programing rate for some well known projects like Safeguard, Computer Systems Command's average programing rate and typical figures like 20,000 statements/MY for (easy) short-term commercial systems a definite functional pattern emerged as shown below:



This relationship is the missing link in the software process. This can be illustrated by removing the cover from the black box:

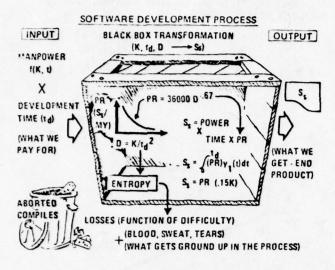


Figure 8. Black Box with cover removed.

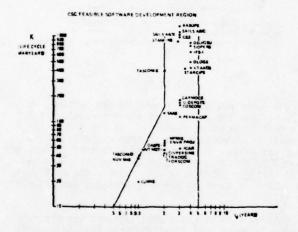


Figure 9. Feasible Software Development Region.

A feasible software development region can be established semi-intuitively. Systems range in size (K MY) from 1 MY to 10,000 MY life cycle. Development times (td) range from a month or two to five or six years. For large systems, the range narrows to 2-5 years for most systems of interest. Five years is limiting from an economic viewpoint. Organizations can't afford to wait more than five years for a system. Two years is the lower limit because the manpower buildup is too great. This can be shown a number of ways.

- 1. Brooks (3) cites Vysottsky's observation that large scale software projects cannot stand more than a 30% per year buildup (presumably after the last year). The Rayleigh equation meeting this criterion is with  $t_d \ge 2$  years.
- 2. The buildup rate invokes Brooks<sup>(3)</sup> intercommunication law, Complexity = N  $(\frac{N-1}{2})$
- 3. Management can't build and control a large software project at rates governed by  $t_{\rm d} < 2$  years without heroic measures.
- 4. Generally, internal rates of manpower generation (which is a function of completion of current work) will not support an extremely rapid buildup on a new project. New hiring can offset this, but that approach has its own shortcomings.

When the feasible region is portrayed in three dimensions by introducing the difficulty,  $K/t_d^2$ , as the third dimension, we have a difficulty surface. Note that as the development time is shortened the difficulty increases dramatically. Systems tend to fall on a set of lines which are the trace of constant magnitude of the difficulty gradient. Each of these lines is characteristic of the software house's ability to do a certain class of work. The constants shown (8, 15, 27) are preliminary. Further work is needed to refine these and develop others.

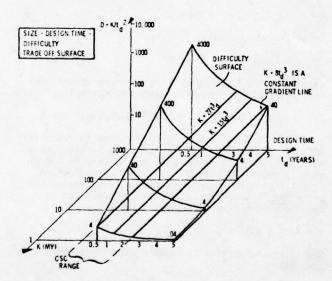


Figure 10. Difficulty Surface.

We can study the rate of change of difficulty by taking the gradient of  ${\bf D}$ .

ii in t<sub>d</sub> direction

jj in K (and \$ cost) direction

grad D points almost completely in the -td direction

The importance of this fact is that shortening the development time (by management guidance, say) dramatically increases the difficulty, usually to the impossible level.

Plotting of all Computer Systems Command systems on the Computer Systems Command systems on the difficulty field shows that they fall on three distinct lines,

<sup>\*</sup> Additional data suggests that Figure 7 may be more complex; i.e., it may be a set of parallel lines, each representative of a certain class of software work.

$$K/t_d^3 = 8$$
,  $K/c_d^3 = 15$  and  $K/t_d^3 = 27$ .

The expression for grad D is:

grad D = 
$$\frac{-2K}{t_d^3}$$
 ii +  $\frac{1}{t_d^2}$  ii (11)

We note that the magnitude of the ii component has this form and the magnitude of  $\mbox{\rm grad}\ \mbox{\rm D}$ 

/grad D/ = 
$$(\partial D/\partial t)^2 + (\partial D/\partial K)^2$$
 (12)

= 
$$3D/3t = 2K/t_d^3$$
, since  $3D/3K << 3D/3t$  (13)

throughout the feasible region.

/grad D/ is the magnitude of the maximum directional derivative. This points in the negative ii (t) direction. The trace of all such points is a line K = constant  $t_d^3$ , or  $K/t_d^3$  =  $C_1$ , or without change in generality,

 $\frac{2K}{t_d}3 = C_2 \tag{14}$ 

Such a line is the maximum difficulty gradient line that the software organization is capable of accomplishing. That is, as system size increases, the development time will also increase so as to remain on the trace of a normal to a constant gradient line defined by

$$K/t_d^3$$
 = C where C can equal either 8, 15 or 27.\*

Study of all Computer Systems Command systems shows that (1) if the system is entirely new--designed and coded from scratch--and if the system has many interfaces and interactions with other systems, C = 8. (2) if the system is a new stand-alone system, C = 15 and (3) if the system is a rebuild or composite built up from existing systems where large parts of the logic and code already exist, then C = 27. (These values may vary slightly from software house to sofware house depending upon the average skill level of the analysts, programers and management. They are in a sense figures of merit or "learning curves" for a software house doing certain classes of work.) The magnitude of these constants, C, is instructive.

 $K/t_d^3$  = C  $\stackrel{.}{=}$  8 applies to new systems that must interact with others within a total MIS structure. This is the toughest task since no logic or code exists and the interfacing problem is formidable. Accordingly, the difficulty gradient for systems of this type will be smaller than for other classes of systems.

 ${\rm K/t_d}^3$  = C  $\pm$  15 applies to new stand-alone systems. These are naturally simpler because the interface problem with other systems is eliminated.

 $K/t_d^3 = C = 27$  applies to rebuilt systems or composite systems where large segments of existing code and logic exist. The main task is integration, interfacing and minor enhancement. Here the entropy is smaller because a significant portion of the disorderedness has been removed by prior work.

\* These constants are preliminary. It is very likely that 6 to 8 others will emerge as data becomes available for different classes of systems.

The significance of grad D is shown in the next few figures. Note that grad D is a vector which points very close to the negative time direction throughout the feasible region.

It is instructive to see how grad D changes as a result of management decisions. We can study this by using some reasonable numbers in the expression for grad D. The first thing we observe is the relative importance of the components.

Assume a typical size system:

K = 400 MY 
$$t_d$$
 = 3 years  
grad D =  $i - \frac{2(400)}{(3)^3} + j\frac{1}{(3)^2} = i - \frac{800}{27} + j\frac{1}{9}$   
=  $i (-30) + j (.11)$ 

#### Importance of components

Magnitude 
$$\frac{i}{j} = \frac{30}{0.11} = \frac{273}{1}$$

The i component (in the time direction) is several hundred times more important.

When the manpower is cut, we get this effect:

Assume a hypothetical management decision to cut the life cycle cost of our typical system by 10%.

Now, K = 
$$(400-40)$$
 = 360 MY.  
grad D =  $i - \frac{-2(360)}{(3)^3} + j \frac{1}{(3)^2}$   
=  $i (-27) + j (.11)$ 

A small change of +3

But all in the time direction. This says we assume the difficulty is less than it really is. The impact will be less product.

No change

The more common case is attempted time compression. Assume that management says instead "Your budget is limited to 400 MY, but we need the system in 2.5 years instead of 3 years".

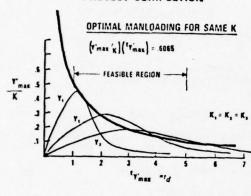
Now K = 400, 
$$t_d = 2.5$$
 years  
grad D =  $i - \frac{2(400)}{(2.5)^3} + j \frac{1}{(2.5)^2}$   
=  $i (-51) + j (.16)$ 

The (difficulty gradient) increased in size from  $30\ \text{to}\ 51.$ 

The result is that shortening the natural development time will dramatically increase the system difficulty.

The risk analysis for project completion is probably the most important aspect of any software system analysis. This is because the entire process is extremely sensitive to small changes in time. Why this is so was just demonstrated in terms of the difficulty gradient. Note, however, in the figure below that shortening the development time by 1/2 year (assuming that we know the job should take 3 years) lowers the probability of successful completion of development to the 3% probability level. This is practically impossible and will guarantee a slip. Brooks (3) points out that adding more manpower will not compensate for too little time. Brooks' statement is equivalent to trying to manload the project according to curve Y2 when it is known that Y3 is the proper curve.

#### PROJECT COMPLETION



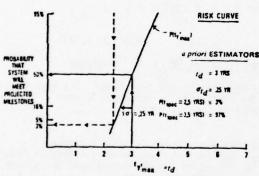


Figure 11. Risk Analysis for Project Completion.

No mention has been made of how to determine the parameters K and  $\mathbf{t_d}$  -- either for a new system or for a system already under way. Space does not permit this development here. Suffice it to say that good empirical methods have been developed to do this and a sound theoretic basis is well along in development. For information on this aspect of the problem, the reader is referred to the references at the end of the article.

#### SUMMARY:

In summary, software systems follows a life cycle pattern. These systems are well described by the Rayleigh (manpower) equation,  $y'=2Kate^{-at\,2}$ , and its related forms. The system has three fundamental parameters: the life cycle effort (K), the development time (t<sub>d</sub>) and the difficulty (K/t<sub>d</sub><sup>2</sup>). Systems tend to fall on a line normal to a constant difficulty

gradient. The magnitude of the difficulty gradient will depend on the inherent entropy of the system. New systems of a given size that interact with other systems will have the greatest entropy and take longer to develop. Rebuilds of old systems, or composites where large portions of the logic and rode have already been developed (and hence have reduced the entropy) will take the least time to develop. New stand-alone systems will fall in between.

The software development region is a difficulty field. Each point is uniquely defined by the three Rayleigh parameters, K,  $t_d$ , and  $K/t_d^2$ .

Systems seek their natural parameters and are inherently stable. This is along a line normal to a \*constant gradient.

The management implications of these concepts are:

Management cannot diminish the development time of a system without increasing the difficulty. All changes take place in the negative time direction. Development time is the most sensitive parameter. It cannot be set arbitrarily by management.

 $_2\text{Given}$  a proper set of project parameters, K,  $t_d$  , K/t $_d$  , a system can be designed to cost with only a small uncertainty.

#### REFERENCES

- 1. Norden, Peter V., "Useful Tools for Project Management", Management of Production, M.K. Starr (Ed) Penguin Books, Inc., Baltimore, MD., 1970, pp 71-101.
- 2. Putnam, Lawrence H., "A Macro-estimating Methodology for Software Development", Digest of Papers, Fall COMPCON '76, Thirteenth IEEE Computer Society International Conference, September 1976, pp. 138-143
- 3. Brooks, Frederick P., Excerpts from The Mythical Man-Month; Essays on Software Engineering, DATAMATION, December 1974.
- 4. Putnam, Lawrence H., A General Solution to the Software Sizing and Estimating Problem, as presented at the Life Cycle Management Conference, American Institute of Industrial Engineers, Washington, D.C., February 8, 1977.
- 5. Putnam, Lawrence H., "ADP Resource Estimating:
  "A Macro-Level Forecasting Methodology for Software
  Development," Proceedings of the Fifteenth Annual
  US Army Operations Research Symposium, 26-29 October
  1976, Fort Lee, VA, Volume I, pp. 323-337.
- 6. Box, George E.P., Putnam, Lawrence H., and Pallesen, Lars, Software Budgeting Model, Mathematics Research Center, University of Wisconsin, Madison (Feb. 1977 Prepublication draft).

## Evolution Dynamics - A Phenomenology of Software Maintenance

M. M. Lehman

Department of Computing and Control Imperial College of Science and Technology London SW7 2BZ

Abstract. The dynamics of evolution of large software systems has been extensively studied in recent years. The present paper briefly reviews the phenomena observed and the main conclusions that follow. The reader is referred to the most recent references and to other contributions to the present Life Cycle Management Workshop for the data and derived models on which these conclusions and generalizations are based. Taken together, these results constitute a phenomenology that provides an essential ingredient of developing computer science.

Key Words. Phenomenology, Evolution Dynamics, Software Maintenance, Programming Process, Programming Management, Life Cycle Management.

#### 1. Phenomenology

The <u>dynamics of evolution</u> of large programs has been discussed in a number of readily available publications, most recently in (BEL76), (LEH76), and (SLC77). There is, therefore, no reason to detail the results or the data on which they are based once again. The interested workshop participant can refer to these papers and to the earlier publications referenced in them. More recent results extending the data can be found in a thesis by Chong (CHO77) and in Patterson (SLC77).

The present contribution to the SLCM Workshop is, therefore, restricted to a restatement of the most general conclusions since it is they that suggest the existence of software phenomenology, a basis for a software engineering science. And phenomenology is one view of the theme of this workshop. We note parenthetically that analogous observations also exist for other computer and computing-related activities leading more generally to a concept of computing phenomenology as a basis for computer science.

The most fundamental implication of our observations is the existence of deterministic, measurable regularity in the life cycle of an application program or of a software system. It has been formalized in the third of our laws of evolution dynamics (figure 1). The observation contradicts a basic assumption normally implicitly accepted by managers of industrial and other large-scale human activity. These generally assume that the progress and rate of progress of a (software) project and the fate of its product, (the software system) is primarily the consequence of management decisions and resource investment. Our conclusion, based on the measurement of a number of systems, projects and organizations, has had to be that in a software development and maintenance environment, product, process and organizational parameters develop during the early stages of the life cycle. Increasingly these then dominate the further life of the system during the continuing maintenance process, producing its characteristic features.

Norden (SLC77) and Putnam (SLC77) reach a similar conclusion from quite a different set of observations.

The form and value to which these process descriptors converge are functions of many factors relating to the nature and history of the product and organization. Precise determinants are not yet understood. But system analysis indicates that the multi-loop feedback nature (checks, balances and controls, for example) of system development and organizational management, is a major contributor to the long-term stability of these parameters, probably in part also of their actual distribution and value. Riordon's Studies (SLC77) supports this by constructing continuous system models whose behavior can be made to reproduce that of various processes by suitable adjustment of their parameters.

#### 2. Evolution

Our first law asserts that large computer programs, software systems that are used, undergo a continuing process of evolution; repair, modification, enhancement, adaptation. The meaning of large in this connection is discussed elsewhere (BEL77) as are the underlying reasons for this continuous change (BEL76), (LEH76). The actuality of the phenomenon has been confirmed and sized by measurement on some eight systems.

## Law of Continuing Change

A large program that is used undergoes continuing change until it is judged more cost-effective to recreate it.

## II Law of Increasing Entrophy

The unstructuredness (entropy) of a program (system) increases with age (change) unless work is specifically invested to maintain or reduce it.

## III Law of Statistically Smooth Change

Measures of global system and project attributes may appear stochastic locally in time and space, but are cyclically self-regulating, with statistically identifiable invariances and well-defined long range trends.

Figure 1. The Laws of Software Evolution Dynamics

It must be stressed that this evolutionary trend is not primarily due to human imperfection in identifying an application and its needs, planning a system and implementing it.

Nor is it mainly due to the underdeveloped nature of many aspects of the programming process, its technologies and its support tools; though the absence of effective structured specification methodologies and software interface control technologies undoubtedly contributes significantly to the magnitude of change in a number of ways.

No! Continuing evolution is intrinsic to the very nature of software and of the use of computing. It is the visible sign of continuing interaction between the system and its environment. Any program is a model of an activity, a process or an aspect of the world in which we live. Or it may be a tool to interact with and modify some part of that world. But the world itself changes continuously, driven by a variety of natural and human, sociological, economic and technological forces. Hence the computer models, the programs, must also continuously change. A program will require maintenance even if -- and this rarely if ever occurs -- its first implementation was perfectly conceived, perfectly designed and perfectly implemented.

## 3. Complexity

As a program is changed, its complexity will generally increase unless some part of the change effort is specifically dedicated to complexity control or reduction. Once again, we cannot here repeat detailed arguments or the data that supports our conclusion (BEL76), (LEH76). We should, however, observe that the second law may be considered an analogue, perhaps even an instance, of the second law of thermodynamics.

Belady (SLC77) discusses various meanings that have been attached to complexity. He reviews proposals for that view of complexity which relates to software maintenance and change. In that context, the concept of complexity may be seen as a measure of the difficulty of implementing a change totally, that is correctly and completely. It may be thought of as reflecting the resistance to change.

Note that even this statement is not unambiguous. There are clearly many separate concerns; complexity of the problem to be solved, of the statement of requirements,

of the consequent system specification, of the procedures to be implemented, of the sub-sequent design, of the system implementation, of the system in use, and of the system maintenance. The implementor's view of the system as he tries to determine precisely what needs to be done to complete a change, for example, will be quite different to that of the user who has to understand how the change will effect his usage in terms of his required actions and the system's response.

Our observations have convinced us that phenomenology is a valid technique for software process analysis. This suggests that despite the many aspects of complexity there are, from a global viewpoint, quantitative relationships between them. This is compatible with our interpretation of the complexity of a system, as measuring the difficulty of comprehending the system from the particular point of one's interest or concern. Some tentative measures have been suggested (SLC77), others are being investigated (LEH77). They all relate in some way to the internal interconnectivity or interdependency of system elements.

## 4. Regularity

### 4.1 The Source of Data

In all of the above we have implied and <u>assumed</u> the existence of a software phenomenology. This is a generalization which, if correct, provides the base for a science of software life-cycle studies and for the engineering and management of software and of the software development, change and adaptation process over the lifetime of an application or a system.

The data on which our observations and conclusions are based, comes from systems ranging in age from 3 to 10 years and having been made available to users in from ten to over fifty releases. They varied widely in their application areas, functional scope, size, in the sophistication of the related process, and the implementation and run-time environments. Yet certain behavioral features appear again and again. It appears justifiable to conclude that they represent basic properties of the software life process, but reflecting also aspects of the methodologies used and of the individual

organizational environments. We hypothesize that many of the characteristics relate to the need for all participants in the creation and use of the system to communicate with one another, and for the system elements (subsystems) to communicate with each other.

## 4.2 The First Law - Continuing Change

We have already discussed continuing growth. In most cases, such growth is at a declining growth rate. And it is possible to show that the decline is not due to decreased pressure for growth; that is a decrease in the demand for additional capability to be added to the system. Nor is it due to a withdrawal of resources from the maintenance process. We can point to many instances where the growth rate declined whilst the people applied and the list of work waiting to be done actually increased (BRO75). Thus the decline is not due to a positive management decision. It stems from the increasing difficulty of working on the system, the fact that as the system ages an ever greater effort is required to achieve unit growth. It is due to increasing complexity, confirmed by the fact that a concerted effort directed to cleaning up a system and reducing its complexity is followed by a period during which the growth rate reverts to its high initial value before settling back on to the main growth path. The increasing complexity has been measured in some of the systems observed, though so far only indirectly.

## 4.3 Complexity

The source of complexity growth lies in the multifaceted nature of any change activity. In particular, apart from the functional change to be achieved, it may be required to complete the change in the shortest possible time, with minimum expenditure, minimum performance degradation, maximum performance improvement (however measured), minimum effort (manpower), minimum effort (intellectual), minimum risk (errors), minimal structural change and so on.

Clearly these subgoals are likely to be incompatible. They cannot possibly all be simultaneously attained from the point of view of any individual change. In terms of the individual manager's interests, structural degradation comes low in the list of priorities — if it is considered at all. Its debilitating consequences, the benefits of attention to its avoidance lie too far in the future. And in any case the degradation due to any one

change is likely to be small -- a GOTO statement to provide access to changed or additional code; a new variable for temporary storage; one more access to some global variable.

The impact of structural decay lies in the future, under commonly adopted standards it does not affect the acceptability of an individual change. But growth of system complexity is the compounded sum of dozens, even hundreds of "inconsequential" increases in system-internal linkages. Clearly our conclusion must be -- strictly control the structural aspects of change or periodically clean up and restructure or both.

### 4.4 Cyclicity and Long Range Trends

One of the most striking features of the data examined is its tendency to regularity. In the simplest case, there is a cyclic variation about some long range trend. That is alternate values of the system or process measure -- taken say at each system release -- lie alternately above and below a trend curve. More accurately, one should say that the average values of the cyclically varying parameter define or determine a long-range trend. In other instances, the pattern is more complex but still identifiably regular and defining a statistically determinable trend.

The cyclicity is interpreted as a consequence of process feedbacks. It is the visible evidence that the programming process and its management organization operate as a feedback-controlled self-stabilizing system. The trend curves and the invariances to be described reflect the long-term stability of these systems.

As local management, which is short-term oriented, deviates from the trend, for example by increasing the rate of work above that historically maintainable, forces develop that sooner or later cause the rate to fall thereby maintaining the long-range trend. In the above example, for instance, the excess work rate would probably cause neglect of or inaccuracy in the documentation. Equally, a period of excessive effort is likely to cause an increased rate of structural deterioration and a larger number of errors. Any one of these factors is self-evidently sufficient to subsequently cause a significant decline in the work rate.

### 4.5 The Long Range Trends

In any large project and large organization there will be many such related effects. It is their sum total over time that determines the net time pattern, the system life cycle. The surprising fact that our investigations and those of Putnam (SLC77) have uncovered is that this net effect is not stochastic in time reflecting sequences of locally inspired, superficially unrelated decisions.

The net effect as expressed by the long range trends can be identified by significant fits to the data. The resultant models can be, and have been, used for project and system planning, committment and project control, to produce for example, a (lifetime) minimum cost project or one that meets its functional, time, and cost goals.

### 4.6 Invariance

Of all the phenomena measured and observed, that of invariance is perhaps the most striking and the most significant, particularly in view of the fundamental role that invariances play in modern physics for example. Several such quantities have been identified. In particular, we have observed that for each of the systems for which sufficient data has been obtained, the work input or effort expended per unit-time has remained constant over the lifetime of the system.

For four of the systems stemming from three organizations, a count of the number of modules changed in each release and the length of each release interval was available. Examing this data as a function of time (system age) -- cumulatively to eliminate the effects of release overlap and slippages -- yields a trend that can only be considered linear. An illustration is given by Lehman and Patterson (SLC 77, figure 5).

This unexpected result, and its occurrence in systems of such widely different environments, is as remarkable as it was unexpected -- and unknown to the managements of and the participants in the various activities. The constant rate of work input, occurred in each case despite advancing technology, increasing tool support, changing language and experience levels and, above all, changing resource allocation to the activity -- both increasing and decreasing. We also point out that this constant work rate does

not imply and did not yield constant work output. Increasing complexity already discussed takes care of that. The output per unit input declines as the system ages.

The insensitivity of the work rate to resources applied, particularly manpower, suggests that many large projects may be operating in a saturation mode. All participants are certainly kept busy — keep each other busy — but the system could also have been built and maintained by a much smaller team. A note of caution must follow. The saturated mode of operation, once started, is likely to have profound effects on the nature, structure and content of the design, code and documentation. These remain reflected in the system throughout its life cycle unless deliberately and consciously removed. Thus, only a major restructuring or recreation effort can permit the switch to a much smaller team.

### 5. The Phenomenology of Management

The underlying cause of the long term invariance of the work input rate is found in the feedback nature of the programming process as controlled by the organization through its management hierarchy. In particular the available data suggests that in large organizations management reacts rather than initiates. Resources are applied to methodology or tool development, for example, in <u>response</u> to the observation of a declining production rate. After all such investment is basically anti-regressive (LEH74), it does not contribute to immediate profitability, to the production of a marketable item.

That is, we deduce from the data discussed that organizational response to declining real output or to the experiencing of productivity, quality and performance problems is to invest resources only to the point where the status quo has been restored. If in exceptional circumstances process development is continued beyond that point, side effects develop that prevent the full and visible return on that investment from being obtained. The system and organization has locked in, stabilized on a handle rate that dominates its further development. Similar interpretations can be placed on the other phenomena observed; that is, on the various short term cyclcities and long term trends.

Why it should be the handle or change rate that is invariant is not clear at this time. Nor is it certain except in outline why we should observe absolute invariance rather than an increasing or a decreasing trend. Thus, these first phenomenological observations must now be followed by analysis and modeling efforts that will reveal the precise structure and characteristics of the feedback systems. Riordon's work (SLC77) represents a significant step in that direction.

### 6. Acknowledgment

While I must accept full responsibility for the opinions expressed in this paper, I wish to gratefully acknowledge the very substantive contribution made during many discussions by L.A. Belady, B. Connell, A.L. Lim, F.N. Parr, J. Patterson, L. Putnam, S. Riordon, and my colleagues on IFIP WG2-3.

### 7. References

- (BEL76) L. A. Belady and M. M. Lehman "A Model of Large Program Development," IBM Systems Journal, Vol. 15, No. 3, 1976, pp 225-256.
- (BEL77) L. A. Belady and M. M. Lehman "Characteristics of Large Systems,"
  Part 1, Chapter 3 in "Research Directions in Software Technology,"
  Published by the MIT Press, Sponsored by the Tri Services Committee
  DOD, Fall 1977.
- (BRO75) F. Brooks "The Mythical Man-Month," Addison Wesley, Reading Mass 1975.
- (CHO77) C. K. S. Chong Hok Yuen "Evolution Dynamics and Its Application to Two Large Programmes," BSc Project Report, Department of Computing and Control, Imperial College of Science and Technology, London SW7, June 1977.
- (LEH74) M. M. Lehman "Programs, Cities and Students Limits to Growth," Inaugural Lecture, 14 May 1974, Published in ICST Inaugural Lecture Series, Vol. 9, 1970-1974, pp 211-229.

- (LEH76) M. M. Lehman and F. N. Parr "Program Evolution and Its Impact on Software Engineering," Proceedings of the Second International Conference on Software Engineering, San Francisco, October 1970, pp 350-357.
- (LEH77) M. M. Lehman "Complexity and Complexity Change of a Large Applications Programme," ERO Research Proposal, March 1977.
- (SLC77) "Software Phenomenology Working Papers and Discussions of the Software Life Cycle Workshop" This volume.

#### PRELIMINARY CCSS - SYSTEM ANALYSIS

Using Techniques of Evolution Dynamics

Meir M. Lehman Imperial College of Science & Technology London

John K. Patterson ALMSA St. Louis

# Abstract

A preliminary study of ALMSA's CCSS logistics system, one of the largest business systems in the U.S., was undertaken in order to compare its evolution with that of the systems described by Belady and Lehman. To some degree a similarity of behaviour is observed but there are also significant differences. The study is less than two months old so present results must be considered very preliminary.

# 1. Introduction

A reorganization of the U.S. Army in 1962 had as a major goal the maximum economic use of data processing equipment. This goal led to the formation of DARCOM's Central System Design Activities (CSDA's). The Automated Logisitics Management Systems Activity (ALMSA), the largest of the CSDA's, has designed, developed and implemented one of the largest business systems in the U.S. - the Commodity Command Standard System (CCSS).

CCSS is a nation-wide data processing system to standardise a part of DARCOM's (U.S. Army Materiel Development and Readiness Command, formerly Army Materiel Command) logistics operation using the IBM 360/65 and MVT systems and various peripherals. CCSS, which consists of over 1300 modules, implements the integrated data base concept and usually requires extensive disk files on-line. It runs at some six separate installations at one of which, for example, one file alone occupies eleven IBM 3330-type disks.

The CCSS project was initiated some ten years ago. Its first release was installed at the first user site in 1972. During the last years continuous pressure has been applied in order to expand new design effort. However, there never seemed to be enough resources to both plan new design and maintain the old. This condition frequently meant that new design was sacrificed [7].

CCSS management and technical personnel have recently had their attention drawn to the Belady-Lehman studies [3]. Superficially at least it appeared that the system and project evolution behaviour reported was rather similar to that experienced within ALMSA. Moreover it was realised that a large amount of data of the type used by Belady and Lehman in their studies was in fact available to describe CCSS history. Hence it was decided to undertake an Evolution Dynamics study of CCSS, both to compare experiences and to see if planning and management tools similar to those reported, could be developed.

This working paper describes preliminary observations. The reader should remember that the study is less than two months old.

### 2. Background

Since this discussion is based on work by Belady, Lehman and Parr a brief summary is in order:

Observations on the development of OS/360 over its many releases for example, indicate that system size, complexity, and cost increase with time. The basic assumption of programming evolution dynamics is that it is legitimate, indeed necessary, to view a large program and its maintenance organisation as interacting systems. Thus one must search "for models that represent laws that govern the dynamic behaviour of the metasystem of organisation, people, and program material involved in the creation and maintenance process" [3]. Field feedback is basic to the process since the system and its designers are being considered as a metasystem. For example, as pressure is exerted to provide bigger releases, the results are more complexity, reduced quality, and other factors which tend to limit the growth rate. Sooner or later this process leads to a release just for restructuring/rewriting. also possible, and has been observed, is an effect called fission where excessive growth leads to system breakup.

The conclusions to date have been summarised in three laws [3], [6]. We are here primarily concerned with the fact that the evolutionary pattern and project attributes of a large-program organisation can be quantised and sized to define cyclic variations, long term trends and invariances. Once identified, these natural system characteristics can be used for planning and management control. Identification of such project and system descriptions forms the first objective of our present study.

# 3. Data Collection and Analysis

Data of the type analysed by Belady, Lehman, Parr and others has been collected for CCSS since its first release. In particular we have available, though not always for all

releases, release numbers and dates, size in modules at release, number of modules effected by the release (handles is the now standard terminology), number of System Change Requests (SCR's) received and closed, number of emergency fixes processed and number of Abends reported. The meaning of these various terms should be clear from the literature and we do not elaborate here.

In the first instance it was necessary to establish a prima facie case for or against the legitimacy of using the evolution dynamics techniques in the CCSS environment. This has been achieved and we now bring some of the first results and preliminary interpretations, comparing them to earlier published results where relevant.

# 4. System Growth

System size has been measured in modules and is shown plotted as a function of Release Sequence Number (fig.1) and of age (fig.2). We observe two cycles of increasing growth rate separated by a shrinkage (clean-up) phenomenon (at some point between releases with sequence numbers 10,14) as described in the references [3],[6]. The two cycles, however stretch over twenty to thirty releases each rather than the three to six described in the references. This may be related to the fact that until recently release interval averaged around 30 days for CCSS with net growth in the range minus forty to fifty modules. This must be contrasted with a release interval range of from three to eighteen months and a net growth of from minus fifty to about four hundred modules for OS/360 in its history up to release twenty.

That is we observe elements of the same growth pattern between CCSS and the other systems, but with differences that we expect to be able to explain in terms of the different operational, usage and maintenance environments.

#### 5. Net Growth

For OS/360 Belady and Lehman argue that a net growth in excess of some 400 modules per release has always led to both delivery and quality problems. They conclude that releases represent stability points in continuous system evolution. Where excessive growth is permitted in a release interval however, stabilisation is not easily obtained, since the change exceeds the intellectual absorbtive ability of personnel and users. A similar conclusion may eventually emerge from the CCSS experience after correlation of figure 2, with the experiences of project participants and users. The first impression is certainly that a growth in excess of some 30-40 modules leads to problems, shrinkage and clean-up. But this observation requires further investigation.

# 6. Complexity

In the earlier studies complexity measures were approximated by the "Fraction-of-Modules-Handled" count. In both systems to which the measure applied an increasing trend with cyclic variation was observed.

In CCSS there is a striking difference, as in fig.4. The fraction-handled in fact, varies cyclically about a constant amount of about 28%. A first attempt at explaining this difference in behaviour suggests that it may be due to a fundamental architectural difference between the systems. For the A and T systems direct communication between all modules of the system is possible and permitted by protocol. Thus as the system ages more and more inter-module connections are established. Hence changes will, in general, tend to spread increasingly across the system; implying, and as implied by, the increasing handle fraction.

CCSS consists of some 150 applications programs each comprising up to about twenty modules. Different programs can communicate with or influence one another only by changing

values in the files to which they have access or by putting requests for further action onto the input queues of the appropriate program or programs. Thus, with hindsight, one should not expect the average fraction of modules handled to change significantly with age. That quantity is, in fact, not appropriate as a complexity measure for a system of this class.

Clearly complexity and its relation to system architecture and class is a matter to be (and actively being) further investigated. But even from the very preliminary observations, important implications to the system designer and software engineer may be drawn.

# 7. Handle Rate

The CCSS handle-rate as a function of system age is shown in figures 5 and 6. The first shows the cumulative (to eliminate release overlap and feedback effects) number of modules handled and yields the, by now familiar, invariance. The average handle rate has remaind constant at about 8 modules per day over a period of more than five years. This fourth instance of work-input rate invariance in fact leads us to identify a fourth law, that of Constant Work-Input. We shall formulate the law more precisely in due course.

The release by release rate shows cyclic variations similar to those previously observed. Careful statistical analysis will be required before any definitive conclusions can be reached.

# 8. Conclusions

The results presented are very preliminary and would not now have been published were they not so relevant to the SLCM workshop. Nevertheless, we have already been able to deduce some managerial guidelines, for example, to plan further releases about the 8 modules per day rate; to restrict growth

to some 30 modules per release <u>or</u> to plan a clean-up release after any in which a growth in excess of this figure was required; and to check any release plan for which the estimate of modules to be handled exceeds some fraction of the system (say between 20% and 40%) as indicated by the cyclic pattern. Our investigations will continue and we hope to report interesting and significant results in due course.

# 9. Acknowledgements

We are grateful to Mr. J. Gilbert and to Colonel Kros for initiating and supporting this study and for permission to publish this initial report. Also to Mr. B. Connell for his active participation in the present study and for his contribution to the analyses.

# 10. References and Bibliography

- 1. Acton F.S., "Analysis of Straight-Line Data", Dover Publications, 1959, pp. 49-50.
- Alberts D.S., "The Economics of Software Quality Assurance", National Computer Conference Proceedings, Vol. 45, 1976, p.441.
- Belady L.A. and Lehman M.M., "A Model of Large Program Development", IBM Systems Journal, Volume 15, No. 3, 1976, pp. 225-252.
- Kosy D.W., "Air Force Command and Control Information Processing in the 1980's: Trends in Software Technology", U.S. Air Force Project RAND, R-1012-PR June, 1974.
- 5. Langley R., "Practical Statistics Simply Explained", Dover Publications, 1970, pp. 160-165.

- Lehman M.M. and Parr F.N., "Program Evolution and Its Impact on Software Engineering", Proceedings of the 2nd International Conference on Software Engineering, Oct. 1976, pp.350-357.
- 7. Lientz B.P., Swanson E.B. and Tompkins L.E., "Characteristics of Application Software Maintenance", ADA 407436, December 1976.
- 8. Tsui F. and Priven L., "Implementation of Quality Control in Software Development", National Computer Conference Proceedings, Volume 45, p.448.

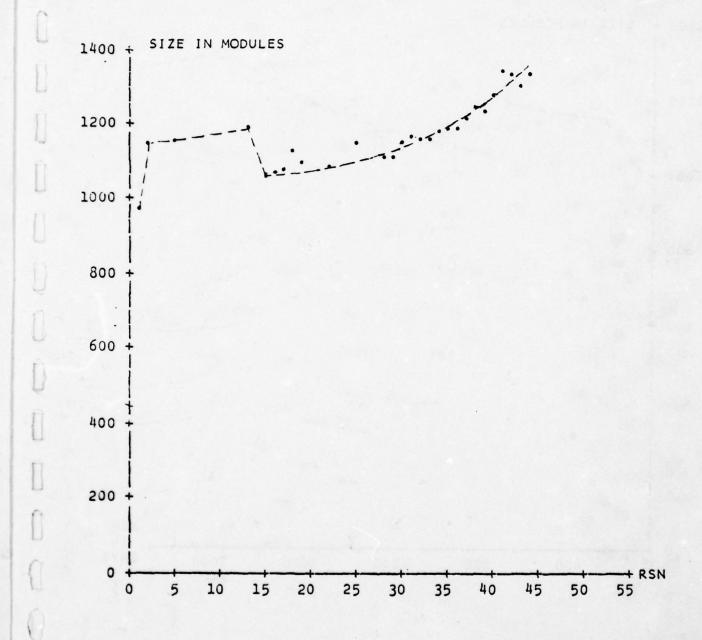


Figure 1: System Size as a Function of Release Sequence Number

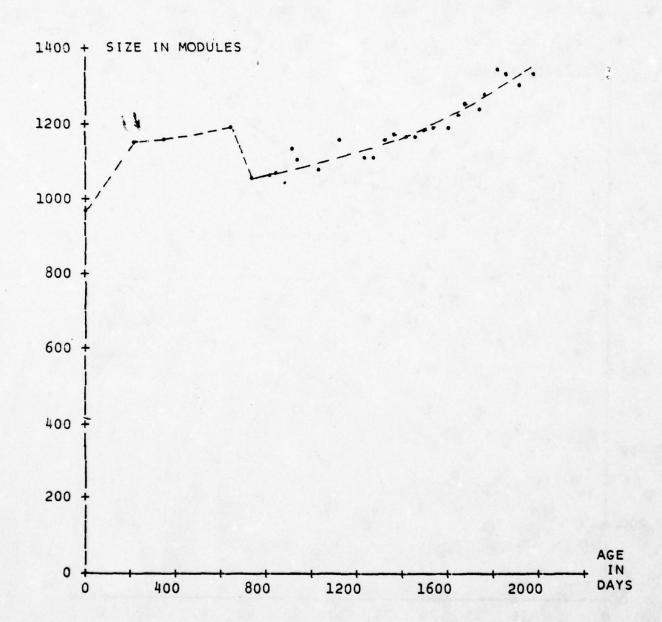


Figure 2: System Size as a Function of Age

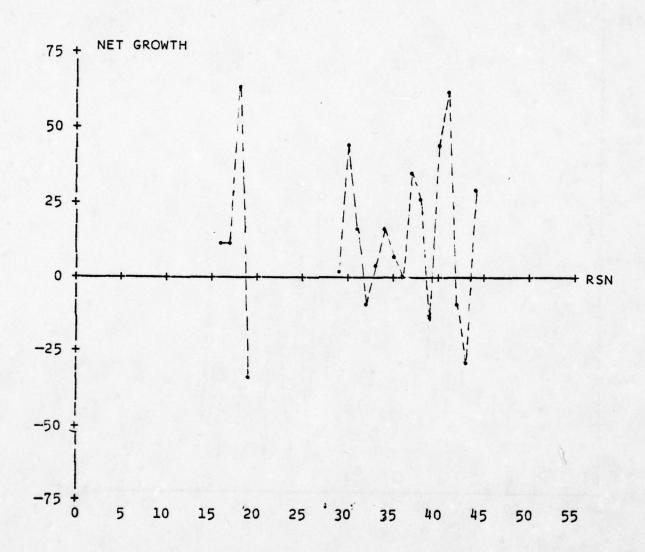


Figure 3: Net Growth by Release

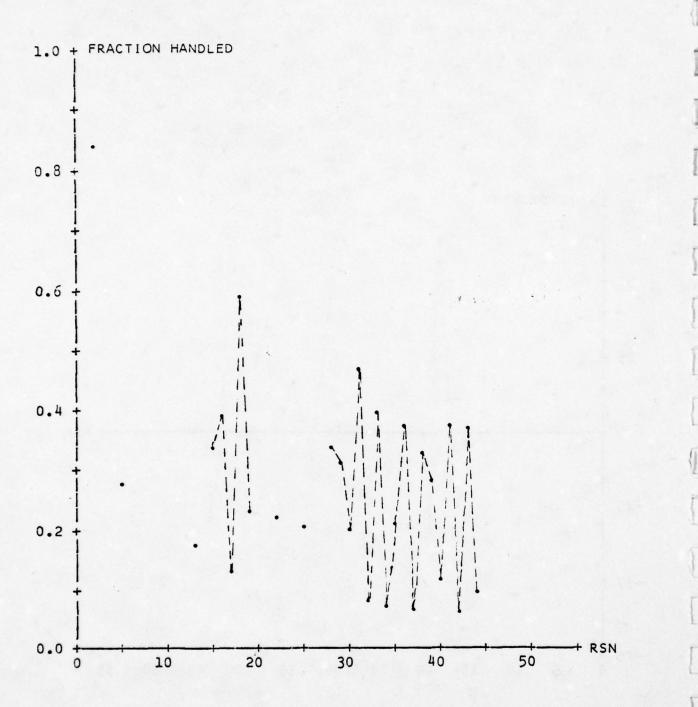


Figure 4: Fraction Handled as a Function of Release Sequence Number

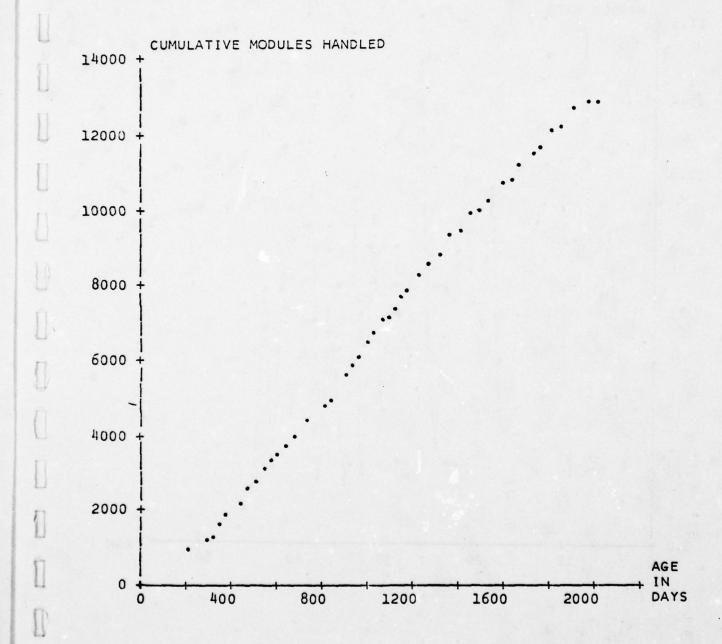


Figure 5: Global Handle-Rate Invariance

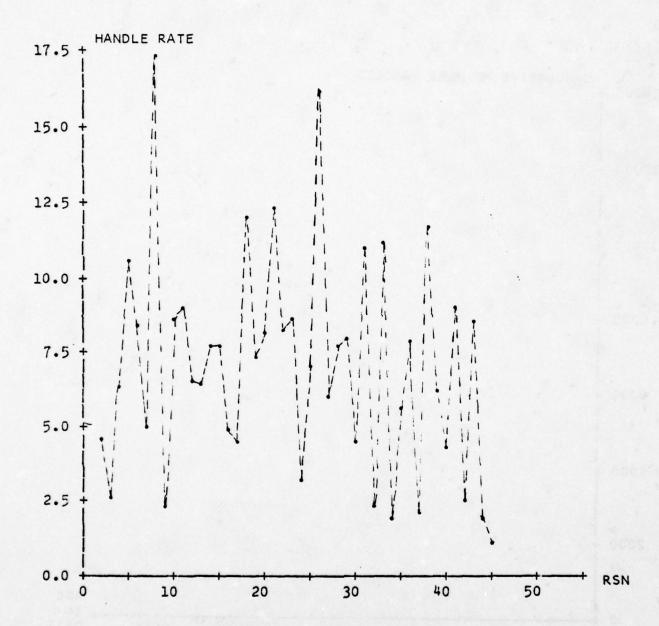


Figure 6: Handle Rate by Release

An Evolution Dynamics Model of Software Systems Development

by

J. S. Riordon

Department of Systems Engineering & Computing Science Carleton University Ottawa, Canada

# ABSTRACT

It has been observed that the life cycle of large software systems typically exhibits several distinct phases. Initially there is a rapid increase in size; following this, growth rate decays, leading to a final phase in which the major effort is directed towards maintenance, with little or no further growth. The last phase usually indicates end-of-life. Based on the concepts of evolution dynamics postulated by Belady and Lehman, the paper develops a fifth-order nonlinear differential equation model of software evolution. The control input takes the form of a cash flow which must be allocated between three forms of effort: creation of new software; testing and fault correction; and maintenance in the face of complexity. A numerical example is presented, and a comparison is made with existing growth data from a large software system.

# 1. Introduction

In recent years, Belady and Lehman [1971, 1972a, 1972b] have collected and analysed data relating to the development of a large software system. It has been found that measures of system activity such as number of modules handled, inter-release time, and total number of modules in the system, show a high degree of regularity which could not be explained on the basis of simple management decisions. The results of this work have been summarized by Lehman [1974]. The basic premise of the research is that a large project of this nature contains its own dynamics. Two postulates are of key importance in describing the system dynamics:

- (a) Law of increasing entropy: "The entropy of a system increases with time unless specific work is executed to maintain or reduce it"
  [Lehman (1974)]. Because of changes in the environment (e.g., hardware changes, new peripherals, batch/time-sharing interaction, addition and deletion of related software) work must be done simply to maintain the software system; otherwise it will "decay", and sooner or later become inoperable.
- (b) <u>Progressive and antiregressive activity</u>: Progressive activity, in the context of a software project, implies the creation of new code. The existence of new code, according to the law of increasing entropy, makes further demands in the form of antiregressive activities. This latter term includes fault correction, testing activity, and the investment in methodology development to combat the complexity which grows with size.

A further paper by Lehman and Parr [1976] contains an important generalisation in that it presents growth data for two other software systems. Each of the three systems has been developed independently by different firms under different conditions (team size, language used, application). The features exhibited by the new data show a similar regularity which adds credence to the precepts of evolution dynamics.

On the basis of these postulates, several regression models have been developed to fit the observed data. In a more recent report Belady and Lehman [1975] have introduced a description which is in a sense more fundamental than the previous ones. This is a differential equation model which attempts to explain the interaction of progression and antiregressive efforts. While simplistic, it does capture the conflicting resource demands of progressive and antiregressive efforts, which ultimately limit the size of the system. This model is used as a starting point in section 2 of the present paper.

### 2. An Evolution Dynamics Model

Feedback effects are inherent in the dynamics of many systems. In analyzing a software development system, one might begin with the very simple model shown in Fig.1. A certain level of resources (money, manpower, equipment, space, etc.) is available at a given time. Resources may be allocated to the development of new and augmented system capabilities on the one hand, and to fault correction, re-design, and repair on the other hand. The first of these is a progressive effort, the second antiregressive. A dynamic relationship arises because the level of antiregressive effort at any time depends strongly upon the relative progressive/antiregressive resources allocations in the past. For instance, the sacrifice of good programming practice and proper documentation in the aid of immediate high productivity returns later in the form of extensive field fault reports and user dissatisfaction. Lack of attention to careful design results in an inflexible and unwieldy system which cannot be made to grow as planned.

With finite resources available, it is evident that contention exists between the requirements for progressive and antiregressive activities. What are the long term effects of a given allocation policy? What constitutes an optimal policy? These questions, which have yet to be answered, are at the heart of evolution dynamics.

### 2.1 System Behaviour

Before a general model is considered, it is worth examining the observed behaviour of an actual system. Fig.2, taken from Lehman and Parr [1976] shows the evolution of system size over a period of some ten years. To quote from Lehman and Parr, the system "is typified by its size, by the richness of its function, by the variety of processing hardware devices and system configurations supported, by the almost infinite variety of users and application environments it serves, by its major use of assembly level languages, and by the size and geographic dispersion of development and maintenance teams."

Two salient features are apparent in Fig.2. The first is the common phenomenon of a trend towards reduced growth rate as time progresses. The second is more unusual: an oscillation, beginning when the system is about 1700 days old, is superimposed on the trend, so that the system grows and shrinks in size with successive releases. Since the latter is clearly not a planned phenomenon, it is surmised that such behaviour is associated with the basic dynamics of the system rather than the direct efforts of management.

It should be noted that Lehman and Parr have collected a variety of data relating to module handle rate, release interval, etc.; in this paper, however, we shall confine our attention purely to system growth.

### 2.2 Belady-Lehman Model

Belady and Lehman [1975] proposed a model in which activity is of three kinds: progressive P, antiregressive A, and additional work related to system complexity, C. It is hypothesized that the cause of C activity is neglect of A activity. The results of cumulative neglect can be removed only by a temporary increase in A. If the total budget, B, is limited, the result is a temporary decrease in progressive activity P. It is assumed that B, P, A, and C can be measured in cost per unit time.

Police Co.

Property of

Suppose, in addition, that:

 $K = \frac{A}{P}$  represents the inherent A activity required

for each unit of P activity, so that complexity does not grow. m = management factor, the fraction of KP actually dedicated by management to A activity  $(0 \le m \le 1)$ .

A balanced budget implies that at any time

$$B = P + A + C \qquad \dots (1)$$

Also 
$$A = mKP$$
, ... (2)

and it is assumed that cumulative neglect of antiregressive activity gives rise to cost C in the following fashion:

$$C = \int_{0}^{t} (1-m) KP dt \qquad ... (3)$$

A block diagram of the Belady-Lehman model is shown in Fig. 3. The open loop transfer function is

$$\frac{C}{B}(s) = \frac{K (1-m)}{K (1-m)+(1+Km)s}$$
 ... (4)

where X = A + C

s = Laplace operator

It follows that, with initial conditions zero, the response to a step of magnitude B is

$$C(t) = B[1 - exp(-t)]$$
 ... (5)

where 
$$\tau_k = \frac{1 + Km}{K(1-m)}$$
 ... (6)

The model, though simple, captures two important aspects of the dynamics, namely resource sharing between progressive and antiregressive effort (here both A and C may be regarded as antiregressive), and the inevitable rise in cost of complexity, which finally absorbs the total budget and limits further growth.

### 2.3 An Augmented Linear Model

Several additional features are needed to make the Belady-Lehman model approximate reality more closely. In Fig.4, P activity is assumed to represent the creation of new code, A activity the action of fault repair, and C activity the maintenance of a complex system. Parameter P represents cash flow available for new programming. The amount of code generated is  $f_p(P)$ , where the function  $f_p$  may be considered initially as a linear multiplier  $K_p$ . Code creation takes time, of course, and it is assumed that the delay can be modelled as a first order lag with time constant  $t_p$  years. System size is obtained by integration of the resultant code production rate.

If the code production rate at time t is R(t) (in modules per month, for example), then, as in the Belady-Lehman model,  $K_aR(t)$  represents the antiregressive resources (in dollars per month) necessary to stem the growth of complexity. In fact, the resources specified by management are  $f_m(K_aR(t))$ , where  $f_m(x) \le x$ . Moreover, the effect of this effort is delayed, as indicated by time constant  $\tau_a$  in Fig.4. The difference between the required and actual resource flow is shown in Fig.4 as D. The complexity cost arising from this shortfall is assumed to be a function  $f_c$  of the time integral of D(t), subject to a lag time  $\tau_c$ . To retain a linear model we assume for the present that functions  $f_m(.)$  and  $f_c(.)$  are simply linear gains m and  $K_c$  respectively.

The open loop transfer function of this linear system is given by

$$\frac{\chi}{P}(s) = \frac{K_a K_p [K_c (1-m) + (K_c \tau_a + m)s + \tau_c ms^2]}{s(1+\tau_a s)(1+\tau_c s)(1+\tau_p s)} \dots (7)$$

Open loop poles occur at

$$s = 0, -\frac{1}{\tau}, -\frac{1}{\tau}, -\frac{1}{\tau}$$

and zeros at

$$s = \frac{1}{2m\tau} \left[ -(K_c \tau_a + m) + \sqrt{(K_c \tau_a + m)^2 - 4m\tau_c K_c (1-m)} \right] \qquad \dots (8)$$

Of particular interest are the extreme cases

m=0: 
$$\frac{X}{P}(s) = \frac{K_a K_c K_p}{s(1+\tau_c s)(1+\tau_p s)} \dots (9)$$

m=1: 
$$\frac{X}{p}(s) = \frac{K_a K_p [(K_c \tau_a + 1) + \tau_c s]}{(1 + \tau_a s) (1 + \tau_c s) (1 + \tau_p s)} \dots (10)$$

When m=0, no antiregressive work is done, and poor system performance may be expected. This is reflected in equation (9) which indicates that closed loop instability can occur for sufficiently large values of "gain". A simplified analysis with  $\tau_a = \tau_c = \tau_p = \tau$  shows that the critical value of  $K_a K_c K_p$ , denoted K, is given by

$$K = \frac{2}{\tau} \qquad \dots (11)$$

It can be concluded that short time constants aid stability, a result which is intuitively reasonable.

Analysis of (10) shows that much great stability is achieved with m=1. This is not surprising, as management is in this case doing all that it can to repair faults and to reduce complexity. Again, long time constants have a destabilizing effect.

# 2.4 Nonlinear Effects

Since it is amenable to analysis, a linear model can be valuable in yielding insight into system performance. For several reasons, however, the concept of linearity must be abandoned if a more accurate model is desired. First, the system behaviour in Fig. 2 cannot be

reproduced by the model described unless large step changes in  ${}^K_a{}^K_c{}^K_p$ , and perhaps m, occur. Although these parameters may vary with time, there is no reason to assume that the variation takes a catastrophic form.

Second, instability of the linear model implies a growing oscillation such that system size, for instance, becomes negative over certain periods. It is difficult to visualize a software system of negative magnitude.

Finally, there are strong arguments related to the real system which point to nonlinear effects. Let us examine first the relationship between the cash flow P for progressive work (coding), and the quantity of code produced. An arbitrarily large code production rate (we assume that we are dealing with meaningful code) cannot be achieved even with an arbitrarily large cash flow. At some point, saturation occurs such that the introduction of additional manpower results in little or no effect on production rate (it can be argued that an actual decrease in output may occur. Perhaps so; this line of thought has not been pursued, however). As an admittedly crude approximation we can therefore assume that  $f_{D}(.)$  has a small signal gain  $K_{D}$  with a hard limiting characteristic taking effect at output level R<sub>p</sub>, as shown in Fig.5. Note that the coding rate can be negative, i.e., the system size can decrease with time; clearly, this effect occurs in system T (Fig.1). In the context of Fig.4, a negative value of P indicates a budget deficit. If the budget B is fixed, and P is negative, the inference is that system complexity has grown to the point where even the antiregressive activities A and C can no longer be supported; there is then no option but to reduce the total system size.

We now turn our attention to the complexity function  $f_c(.)$ . In the ideal case in which programs are highly structured and fully documented, it may be argued that complexity cost varies linearly with system size. In the worst case, an exponential relationship exists. In this model parameter D represents a remnant of work undone, and the worst case is assumed. Thus, for an input v, the function  $f_v(v)$  is

 $f_c(v) = a \exp (bv)$  ... (12) where a and b are constants.

A third nonlinearity is  $f_m(.)$  in Fig.5. For negative input, the multiplier should be zero, so that a "diode function" is created. This effect was not easily reproduced by the simulation package discussed in section 3, and had to be ignored.

A simulation model of evolution dynamics is shown in Fig.6. System equation are

$$\dot{x}_{1} = x_{2}$$

$$\dot{x}_{2} = \frac{1}{\tau_{c}} \left( -x_{2} - x_{3} + K_{a} x_{4} \right)$$

$$\dot{x}_{3} = \frac{1}{\tau_{a}} \left( -x_{3} + mK_{a} x_{4} \right)$$

$$\dot{x}_{4} = \frac{1}{\tau_{p}} \left[ f_{p} (B - f_{c}(x_{1}) - x_{3} - x_{4}) \right]$$

$$\dot{x}_{5} = x_{4}$$
(13)

# System Simulation

# 3.1 Parameter Values

The dynamics of equation (13) were simulated on an interactive graphics system at Imperial College, London. Each "run" covered a period of ten years; a Runge-Kutta integration routine was used with a step size of 0.2 years. Parameter values listed below are representative only, the intent at this point being to examine the general behaviour of the model, rather than its detailed numerical validity.

 $\tau_a = \tau_c = \tau_p = 0.4 \text{ years}$   $K_p = 100 \text{ modules/megadollar}$   $R_p = 250 \text{ modules/year (saturation level)}$  a = 1.8 b = 0.2complexity factors

# 3.2 Simulation Results

Figs.7-9 show the results associated with several values of input budget rate B and management factor m. State variables plotted are:

 $x_1$ , the argument of complexity function  $f_0(x_1)$ 

x<sub>4</sub>, the code production rate (modules/year)

x<sub>5</sub>, system size (modules)

In Fig.7, the budget B is a step of magnitude \$4M per year, and m=0.7, i.e., 70% of the antiregressive work necessary to hold complexity in check is in fact carried out. The simulation shows that the code production rate (the progressive element) increases to a maximum of about 225 modules/year at the end of the first year. At that time the complexity has increased to the point where such a production rate cannot be sustained within the budget available, since an increasing resource demand is being made by A and C activity. A balanced budget requires a reduction in P activity, which later leads to a reduction in A activity. By year 6, the system size is close to its asymptotic size of 500 modules, and complexity is also nearly constant. It is evident that, with the resources available, the system has reached its limiting size.

Suppose now that it is desired to increase both the coding rate and the final system size. A generous increase is made in cash flow, to \$10M per year. The results are shown in Fig. 8. Coding rate rapidly approaches its saturation level of 250 modules per year, and growth is nearly linear for about 2 1/4 years, the total system size

reaching 490 modules at that point. Complexity has also risen rapidly. Shortly after time  $T_R$ , the resource conflict assumes the form of a crisis. Prior to  $T_R$ , the system had excess manpower (indicated by saturation in the coding rate); now manpower is shifted suddenly away from P towards C activity. Within a further six months growth is at a standstill, and a budget deficit is incurred. To reduce complexity cost, it is then necessary to reduce system size, and an oscillatory pattern sets in. Effort shifts back and forth between progressive and antiregressive (both A and C) activity. Each cycle is characterised by a rapid initial growth whose resulting complexity cannot be handled within the resources available; a partial collapse ensues, complexity diminishes, and the cycle repeats. In this case a mean system size of 680 modules is reached by year 10, with a superimposed oscillation of magnitude  $\pm$  40 modules.

In Fig.9 the exponential complexity factor b is taken as 0.14, rather than 0.2 as previously. This corresponds to the expenditure of additional effort in maintaining well structured software, so that the cost of increasing complexity is less severe than previously. Again, however, the system budget is \$10M per year. Fig.9 shows, however, that the system now continues to grow "gracefully" over the ten year period with an asymptotic value of about 1400 modules.

#### 4. Conclusion

A nonlinear differential equation model of evolution dynamics has been developed, and preliminary analysis and simulation have been carried out. It has been found that the dynamic model is capable of reproducing some important phenomena observed in the data and that, in turn, model parameters can be related to observed characteristics of the actual system.

While the results are promising, a great deal of work must be done before practical results ( in the form of an accurate predictive

model) can be achieved. The first step is the determination of a suitable model structure. The one presented here is oversimplified, in that no account is taken of important variables such as handle rate, inter-release time, and fault rate. In addition, it is incomplete in that it contains no complexity component which is purely a function of system size. It is therefore possible, with the present model, to set m=1 and achieve indefinite growth with negligible complexity cost, a result unlikely to be duplicated in the real world. Indeed, the question of the structure of a complexity model alone is one worthy of intensive effort.

The second step is the determination of realistic system paramter values. The present data is inadequate to use for any form of statistical parameter estimation. It is possible that more detailed estimates of time constants, for instance, can be obtained by the collection of further data. Clearly the structure of the model developed will have a strong influence on data requirements generally.

The third major stage is the application of control theory to the models developed earlier. Providing these are realistic, the "payoff" here can be considerable in financial terms. Consider as an example the system growth curves of Fig.7-9. In Fig.7, some 500 modules were developed and maintained over a ten year period at an average cost of \$80,000 per module (recall that the budget was \$4M per year for ten years). In Fig.8 a mean level of 680 modules was achieved at an average cost of \$147,000 per module. It seems evident that great savings can be made by the correct control strategy. Even if such a strategy cannot be applied in detail (due to stochastic perturbations such as sudden demands of major customers), the results should provide valuable guidelines to management.

An additional line of research, which is probably premature at present, is the application of evolution dynamics principles to a wider group of commercial, industrial and socio-economic systems.

The work described in this paper constitutes a bare beginning of a serious effort to apply feedback modelling techniques to evolution dynamics. However, the results to date indicate quite clearly that the effort is likely to prove rewarding in both academic and industrial terms.

#### References

Belady and Lehman (1971)
Belady, L. A., and Lehman, M. M., "Programming System Dynamics",
IBM Research Report RC3546, September 1971; 30pp.

Belady and Lehman (1972a) Belady, L. A., and Lehman, M. M., "An Introduction to Growth Dynamics", Statistical Computer Performance Evaluation, Academic Press, 1972; pages 503-11.

Belady and Lehman (1972b)
Belady, L. A., and Lehman, M. M., "A Systems Viewpoint of Programming
Projects", Imperial College, Department of Computing and Control Research
Report 72/31, 1972. Also published in Advances in Cybernetics and Systems,
I. Gordon and Breach, London, 1975; pages 15-28.

Belady and Lehman (1975)
Belady, L. A., and Lehman, M. M., "The Evolution Dynamics of Large
Programs: IBM Systems Journal, Vol.15, No.3, 1976.

Lehman and Parr (1976)
Lehman, M. M., and Parr, F. N., "Program Evolution and its Impact on Software Engineering", Proc. 2nd International Conference on Software Engineering, San Francisco, Oct. 1976, pp.350-357.

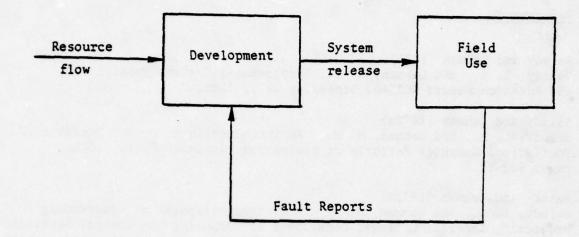
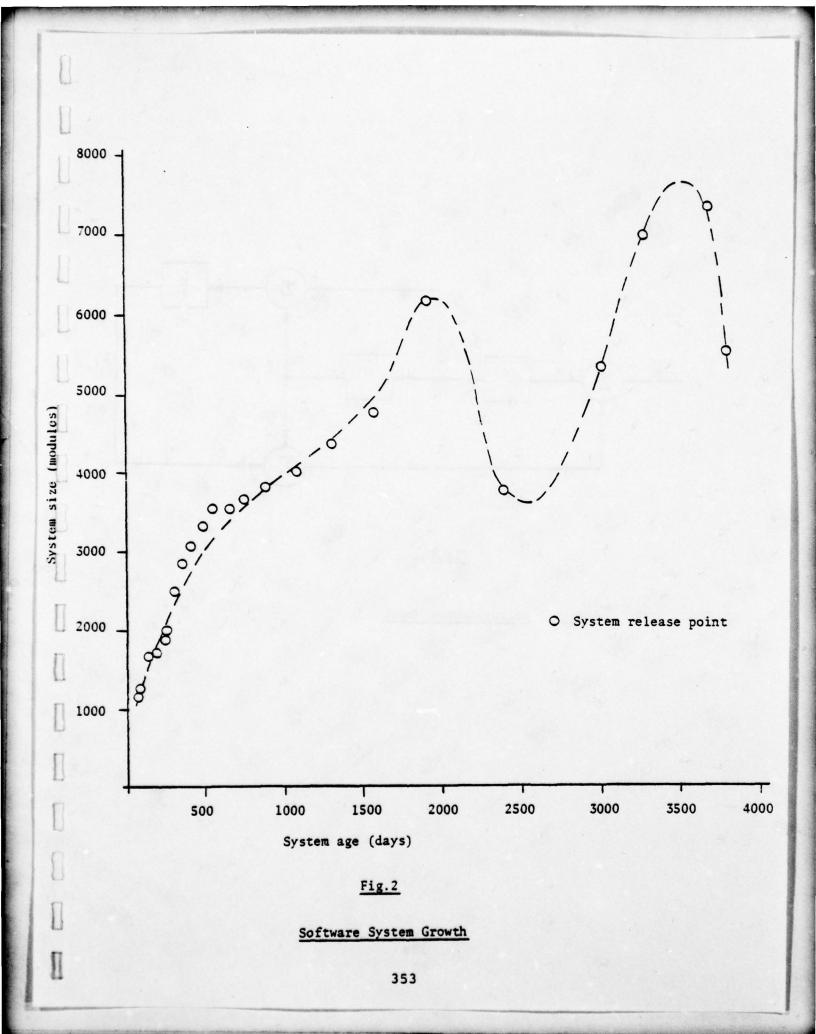


Fig. 1

# Basic Dynamics of System Evolution



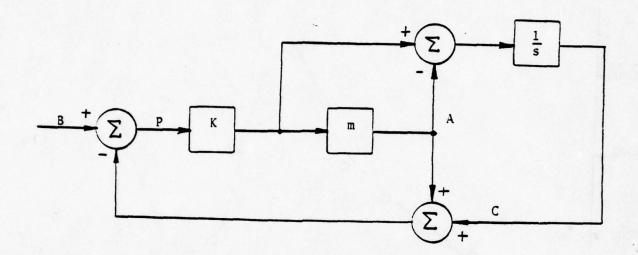
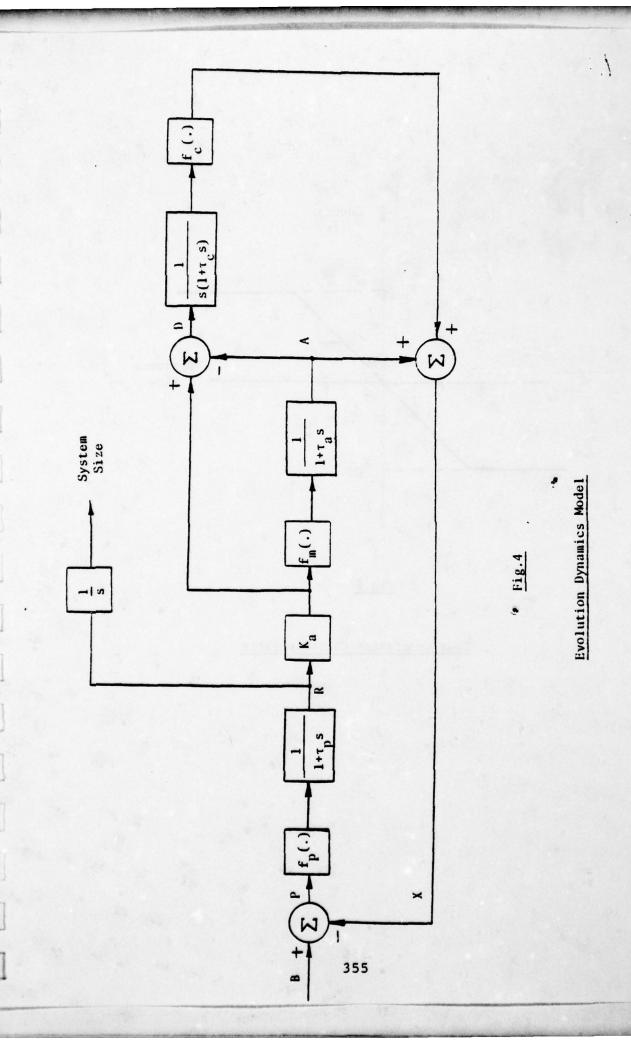
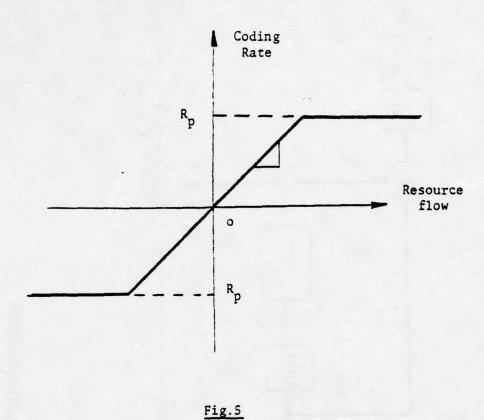


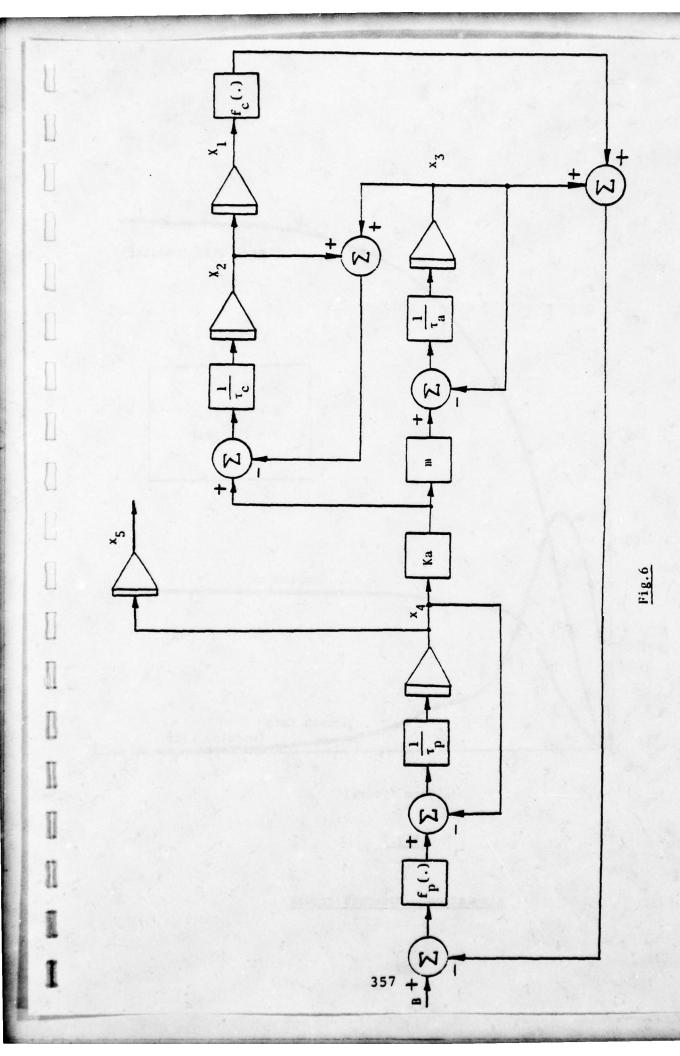
Fig.3

Belady-Lehman Model





Code Generation Characteristic



Evolution Dynamics Simulator

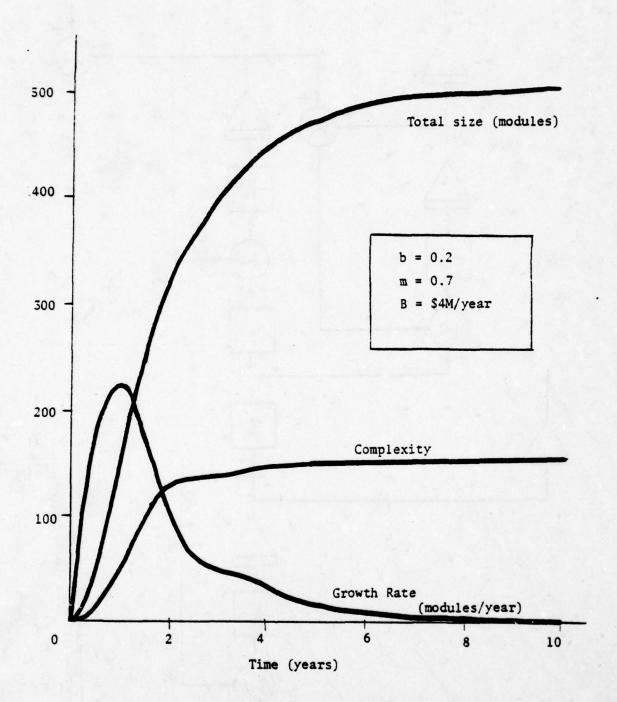


Fig. 7

Simulation: Moderate growth

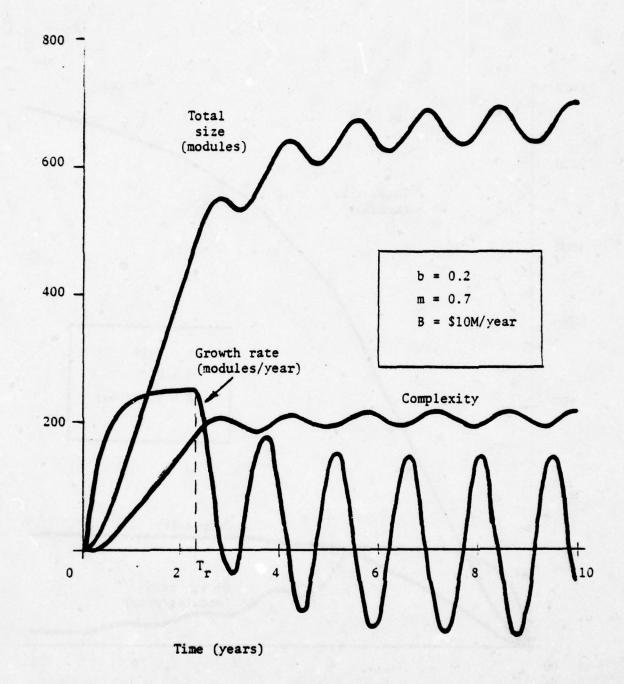
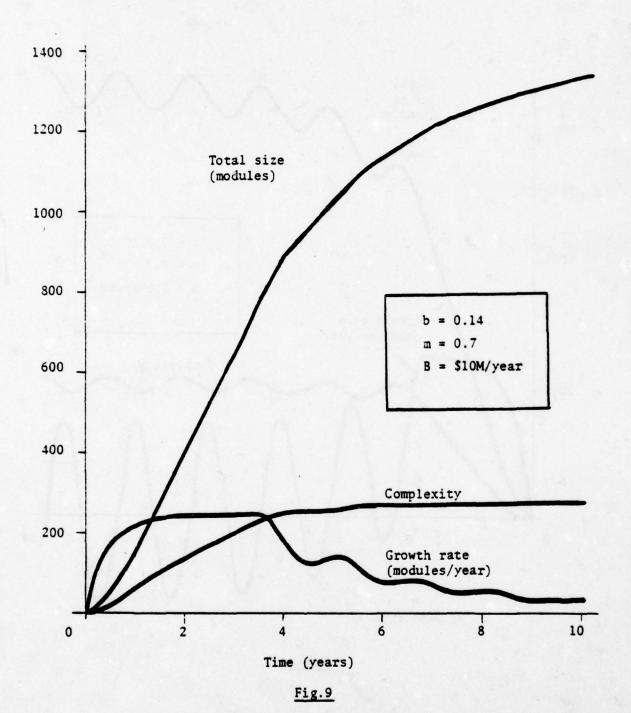


Fig.8

Simulation: Forced growth



Simulation: Reduced complexity cost

#### THE DYNAMICS OF SOFTWARE DEVELOPMENT I

#### GEORGE J. FIX

ABSTRACT: The purpose of this study is to develop a macro-methodology for management that will provide accurate estimates of manpower and time to reach critical milestones of software projects. There are two basic aspects of this subject, namely,

- (1) System dynamics a model that will describe the evolution of the project in time.
- (2) Equation of state for the project the fundamental relationships between input parameters to the dynamic model. This report treats the first subject by developing a differential equation of the process which allows one to model the effects of changes in requirements in the middle of a projects' development. From the project control viewpoint this permits one to quantitatively determine the cost impact of changes to an ongoing development and assess its economic value before implementation.

#### THE DYNAMICS OF SOFTWARE DEVELOPMENT I

#### G. J. FIX

1. <u>INTRODUCTION</u>. Application software development has been an area where standard managerial and cost controls have proven ineffective in many important respects. Brooks [1] and Putnam [2] have pointed out instances of actual costs several times the initial budgeted cost and time to initial operational capability sometimes twice as long as planned.

The purpose of this study is to develop a macro-methodology for management that will provide accurate estimates of manpower, and time to reach critical milestones of software projects. There are two basic aspects of this subject, namely

- (1) System dynamics a model that will describe the evolution of the project in time
- (2) Equation of state for project the fundamental relationships between input parameters to the dynamic model.

In this report we shall treat only the first subject leaving the second to a latter report. The starting point is the software model developed by Putnam [2] which was inspired in part by the work of Norden [3]. We shall show in Section 2 that this model is equivalent to a differential equation involving time and man-year requirements. This equation is not the entire story since it contains parameters which must be supplied through the project's equation of state. However, the differential equation does offer an improvement over the "Norden equation" of [3] in that it covers cases not treated by the latter. For example, it can be used to model the effects of a change in requirements in the middle of the project's development. These and other examples are given in Section 3. The last section contains a description of the computer program used in these simulations.

2. THE DYNAMIC MODEL. Putnam [2] has identified two basic parameters in any software project, namely the time to peak development and the system difficulty D. The total number of required man-years K is related to these parameters by

 $D=K/t_0^2$  (1) In a "Norden type evolution", Putnam has shown that number of man-years at any given time is given by

where p is the learning coefficient. Putnam has suggested that  $p(t;t_d) = +\frac{t}{2t_d}$  (3)

describes software projects quite adequately. Observe that implicit in the choice is an assumption that the system learns (or better improves

its ability to overcome problems in coding, design, etc.) in a linear way with time, i.e.,

 $p(t;t_d)$  - t. That the coefficient p has the specific form (3) follows from the definition of  $t_d$  which we recall is the time where  $\dot{y}(t_d)=0$ .

Putnam has shown that the S shaped "Rayleigh curve" (2) provides an excellent description of the time evolution of y under ideal conditions. It has in addition the attractive feature of a superposition of similar S shaped curves representing coding, design, etc. See Putnam [2].

The relation (2) does have one major disadvantage that limits its usefulness for some software projects, namely it assumes that the system difficulty D is constant throughout. This, for example, does not permit the model to be used for projects where objectives or software requirements are changed before the completion of the project. In the remaining part of this section we shall develop an alternate model which reduces to (2) when D is constant, but can also be used when D is variable.

The idea is to view software development as a dynamic system tending to a steady state, i.e.,

 $y(t) + K=y(\infty)$  as  $t+\infty$ , (4) There are two types of forces acting on the system, one being negative (entropy to be overcome for completion of the project) and the other is positive. The negative force has the form

 $0 - \frac{y(t)}{t^2} \tag{5}$ 

and is the <u>effective system difficulty</u> at time t. As above D is the total system difficulty. It is determined from the projects equation of state, and we admit the possibility that it may change in time (as it will if for example the system's requirements are changed in the midst of the project). We also admit the dependence of D on y, y, i.e.,

D=D(t,y, $\hat{y}$ ), which reflects Brooks' Law [1], namely that the system difficulty will depend on intercommunication and hence increase with y, or say  $\hat{y}$ .

The positive force works against (5) and reflects the ability to learn that is fundamental to Norden's model. In particular it has the form  $p(t;t_d)\frac{dy}{dt}, \qquad \qquad (6)$ 

and increases with the learning coefficient p. A balance of forces gives

 $\frac{d^2y}{dt^2} = -p(t,t_d) \frac{dy}{dt} + D - \frac{y}{t_d^2},$ 

or what is the same,

$$\frac{d^2y}{dt^2} + p(t,t_d) \frac{dy}{dt} + \frac{y}{t_d^2} = D$$
 (7)

with the initial conditions,  $y(0) = \frac{dy}{dt}(0) = 0$ (8)

If we use the linear learning law p t, then (7) becomes

$$\frac{d^2y + t}{dt^2} + \frac{dy}{dt} + \frac{1}{t^2} y = 0$$
 (9)

Observe that if D is a function only of time with  $D(x) = \lim_{x \to 0} D(x)$ .

then the solution y(t) of (8)-(9) will satisfy (4) with  $K=D(\infty)$   $t_d^2$ . Indeed, using variations of parameters we have

$$Y(t) = t_d \int_0^t \exp\left\{+\frac{1}{2} (t/t_d)^2 - \frac{1}{2} (t/t_d)^2\right\} \int_0^T ds D(s)$$
 (10)

The relation (4) following by integrating by parts and then passing to the limit as  $t\rightarrow\infty$ . Also observe that if D is a constant, then

$$Y(t) = t_{d}^{2} \int_{0}^{t} \exp \left\{ + \frac{1}{2} \left( \frac{t}{t_{d}} \right)^{2} - \frac{1}{2} \left( \frac{t}{t_{d}} \right)^{2} \right\} \frac{\tau}{t_{d}} d\tau \quad (11)$$

$$= K \left\{ 1 - \exp \left[ -\frac{1}{2} \left( \frac{t}{t_{d}} \right)^{2} \right] \right\} \quad (12)$$

which is exactly Norden's equation.

3. <u>SOME EXAMPLES</u>. In this section we report examples that illustrate the dynamic model [9, Section 2]. In each case the system difficulty D is a function only of time, and hence [(10), Section 2] is an exact solution, nevertheless it was more convenient to use a numerical solution of (9). The program is described in the next section, and a listing is included for future reference.

Two systems were chosen for purposes of illustration, one large and the other medium sized. The data came from IBM and was communicated to the author by Putnam.

The first example is IBM program No. 30 which has D = 77.73 man-years

and

 $t_d$  = 2.125 years The Norden curves for Y and  $\hat{y}$  are shown in Figures 3-1 and 3-2, respectively. Also plotted in these figures is the dynamic model [9, Section 2] where at time

t =  $t_d/2$  the difficulty was increased by a factor of 1.25. Observe that this resulted in an increased value of K (from roughly 350 to 430) and anincrease in  $t_d$  (from 2.125 to 2.4). Observe also that the effect on the system due to the sudden change in D was not felt instantaneously. The greatest effects were seen roughly a year after the impluse. This shows that even a modest perturbation of the system can have definite effects on large systems, and that these effects may not be apparent until a much later time.

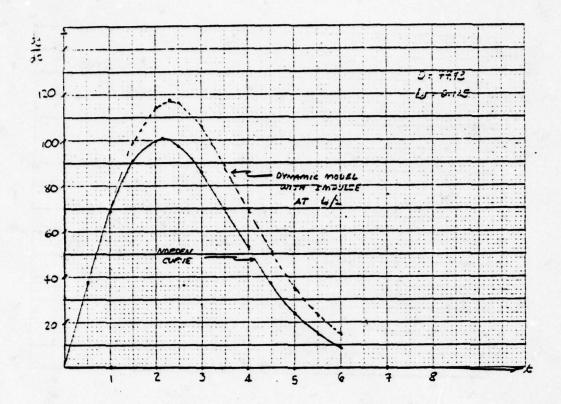


FIGURE 2 IBM PROGRAM NO.30

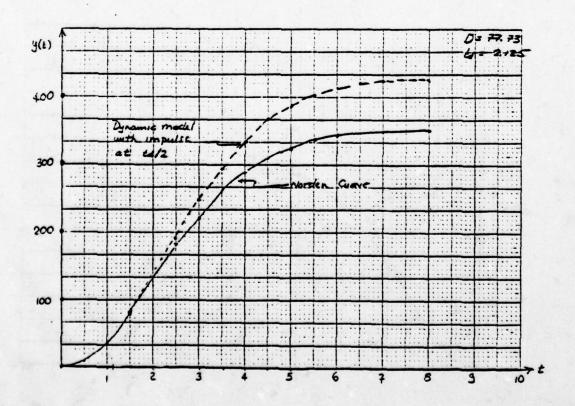


FIGURE 1 IEM PROGRAMAMESO

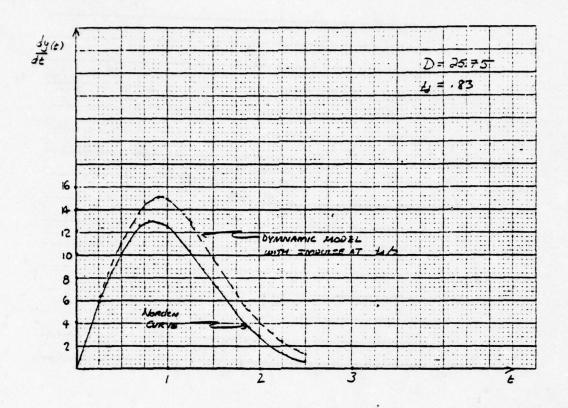


FIGURE 4 IBM PROGRAM NO. 5

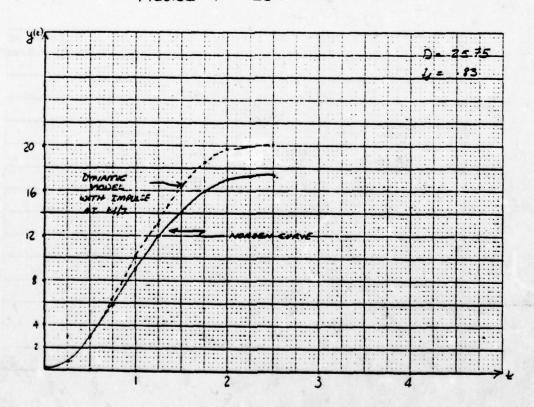


FIGURE 3 IBN PROGRAM #5

A similar experiment was done on the smaller IBM program No. 5

where

$$K = 25.75$$

and

 $t_d = .83.$ 

Observe that the effects of the perturbation are less severe and detectible sooner.

4. Numerical Scheme. We rewrite the differential equation [9, Section 2] as a first order system y = v (1)

$$v = -t_d^2(Y + tv) + D(t,y,v)$$
 (2)

To integrate this equation we introduct a grid

$$0 = t_0. < t < \dots < t_k < \dots$$

with

$$t_k + t_k = \frac{1}{2} (t_k + t_k + 1).$$

Put

$$v_r = v(t_k)$$

Then the approximation is

$$\begin{cases} Y_{k+\frac{1}{2}} = Y_{k} + \Delta t & V_{k} \\ V_{k+\frac{1}{2}} = V_{k} - \Delta t & t_{d} \\ V_{k+\frac{1}{2}} = V_{k} - \Delta t & t_{d} \\ \end{cases} (Y_{k} + t_{k} & V_{k}) + D(t_{k}, Y_{k}, V_{k}) \Delta t \qquad (4)$$

and

$$\int Y_k + 1 = Y_k + \Delta t v_k + \frac{1}{2}$$
 (5)

$$\int v_{k} + 1 = v_{k} - \Delta t \ t_{d}^{-2} (Y_{k} + \frac{1}{2} + t_{k} + \frac{1}{2} v_{k} + \frac{1}{2})$$

$$+ \Delta t \ D (t_{k} + \frac{1}{2}, Y_{k} + \frac{1}{2}, v_{k} + \frac{1}{2})$$
(6)

Where  $\Delta t = t_k + 1 - t_k$ . This is a 2nd order predictor-corrector method, and is of the Runga-Kutta type.

The functions C and XK are the coefficients of  $\mathring{y}$ , Y, (respectively) and FCN is the time dependent system difficulty.

## Bibliography

- 1. Brooks, F.P., "The Mythical Man-Month, DATAMATION, Dec 1974.
- Putnam, L.H., "A General Solution to the Software Sizing and Estimation Problem", Manuscript
- 3. Norden, P.V., "Useful Tools for Project Management" from "Management of Production, M.K. Starr (ed.), 1970.

SOFTWARE COMPLEXITY

L. A. Belady IBM T.J.Watson Research Center Yorktown Heights, N.Y. 10598

August 1977

ABSTRACT: Earlier work on program complexity is briefly reviewed. Observed large scale software evolution and some measures of its complexity are then described. An information theoretical model by Beilner and Belady formulated but unpublished in 1975 is introduced to characterize the three distinguishable components of software modification effort: identification of the impacted modules, coordination of changes among the modules, and finally the implementation of changes. Some results are then related to software design, maintenance and documentation. (This paper was prepared for presentation at the Workshop on Software Life Cycle Management, sponsored by the U.S Army Computer Systems Command, August 22-23, 1977.)

#### INTRODUCTION

It is a natural desire to compactly capture and attempt to measure the agony we encounter while working with software, spanning the entire life cycle from requirements to operation and maintenance. The program in its machinable form is the most obvious manifestation of the reasons for great and often unpredictable cost. Thus, not surprisingly, the majority of measures proposed so far simply summarize some countable parameters of this end product, in the hope of being able to characterize and analyze difficulties of creation, operation and modification of software (GIL) or at least reveal how frequently different software constructs are applied in actual use (ELS), (KNU).

The study of computational complexity <AHO>, for example seeks to find for a given algorithm the cost - processing time or memory space - as a function of problem size, but says little about other properties. A relatively new and related area of research is algorithmic information theory <CHA>. Here for instance one wishes to establish the minimum amount of information necessary to specify an algorithm. Another sought after measure is the probability that a random bit pattern acting as a program produces a particular output and then terminates.

Other work is concerned with identifying descriptive parameters, essentially those which describe the control structure of a finished program (BAS), (MCC), (HAL). Typically, the number of branching points is considered dominant while the contribution of straight line code to complexity is often suppressed. Occasionally the formula for entropy is applied, but rarely explored to its probabilistic foundation.

It seems to be discouragingly difficult to construct a complexity index which measures not only the end product but also the process leading to it. And since it is becoming increasingly evident that the cost of changing existing programs prevails over that of their creation, perhaps quantized phenomena which measure difficulty and consequences of program modification bring us a bit closer to a satisfactory metric.

Three approaches to capture modification complexity are reviewed in this paper. The first one is essentially a measure of diffusion and stratification of changes into the rest of the system as observed in connection with actual large software and later reported (BEL). The second one focuses on the relative locality of declared variables for software expressed in block oriented languages (HOO). Finally, a probabilistic model of programming complexity is offered, leading to the formulation of an entropy measure. Some ideas of information theory are then interpreted in

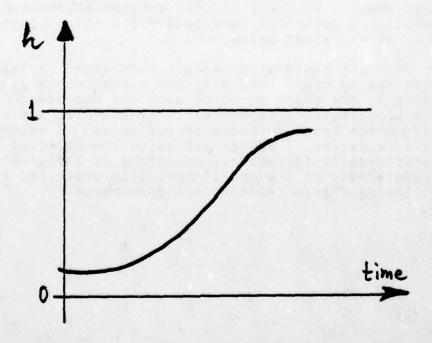
this context and conclusions drawn related to design, documentation and maintenance practices.

COMPLEXITY GROWTH OBSERVED DIFFUSION OF CHANGES AND EXPANSION OF SCOPE

In this paper we consider all software systems made of interdependent modules, not simply a collection. The internal detail of these modules is hidden and will not concern us here. We just observe, and attempt to characterize, the response of this system of modules to a series of changes (error repair, functional enhancement). A module is subjected to a change if its machinable forms before and after the change are not identical. Other characteristics of a change are not examined here.

Large systems typically evolve over a series of so called releases. At each release batches of changes, accumulating over a period of, say, several months, are applied. Examples of this are operating systems and large application software, such as electronic line switching, banking, etc.

Let h be the ratio of changed modules to the total number of modules before the change. On a large operating system, which had been evolving over a decade of about twenty releases, h was observed (BEL) to be monotonically increasing and approaching unity as depicted in the figure below:



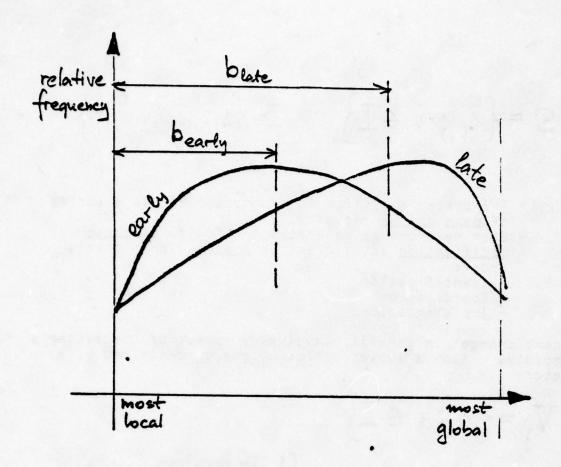
Interestingly, the rate of modification activity over the entire period of observation was verified to be essentially steady state, yet the dispersion of changes displays a faster than linear tendency, though obviously bounded by unity. The spread of changes is thus greater in an old and evidently more unstructured system - an indication of structural aging.

From among many potential system parameters the authors found h to be the closest to an acceptable index of complexity. It is an observed, and directly measurable, quantity which also describes a system property, namely its resistance to change. Indeed, effort spent is likely to be proportional to the number of modules involved in system modification.

The same report (BEL) also offers a model of error repair diffusion. Following this, during a repair interval between two releases some errors are removed, with the net result of leaving residual errors and injecting new ones due to imperfection of the repair process itself. This error flow recursively applied to each interrelease interval displays then an exponential growth of the number of error classes, each class with different process history. This stratification is offered as a major contributor to the deterioration of structure, hence system complexity.

The second approach mentioned in the introduction has its origin in studies performed on a quite differenct software system which was written in Algol 60, a block oriented language <HCO>. In these studies another phenomenon was dicovered as a potential candidate for a novel complexity measure, as explained below.

Due to explicit declaration of all referenced variables, it was possible to count them all, and subsequently group them according to the block at which each was declared. This spans a distribution, so one can, for example, define b, the expected block level or scope of variables. It was noticed that over a period of system evolution the expected scope was monotonically increasing, analogous to the growing unstructuredness of the operating system described above. The following figure depicts this phenomenon.



One of the goals of structured design is to keep variables as local as possible in order to control complexity. However, this becomes increasingly difficult when changes are to be effected which do not fit into the original architecture. Structural clarity unavoidably suffers. Perhaps b and its variation during system evolution are worth while monitoring as complexity indicators during the entire life cycle of large software, whenever disciplined declaration of variables is possible.

# A PROBABILISTIC COMPLEXITY MODEL - INFORMATION THEORY APPLIED

The previous section already implied the two major aspects of software modification complexity: complexity contributed by the content of the change (process) and that by the internal structure (system). In building our model we wish to keep these two contributors to complexity as separate as possible. More formally:

 Let the system under consideration consist of n elements or modules

$$S = \{s_i ; i \in I\}$$
  $I = \{1, 2, ..., i, ..., n\}$ 

- The system evolution is decomposable into a series of <a href="mailto:next change">next change</a> requests.
- Each next change generates the following three modification activities, or phases, and in this order:
  - Identification
  - Coordination
  - Implementation

A next change, in general, involves a subset of the system's n modules. Such a subset called R, can be described by a vector

$$V_j = \{v_i | i \in I\}$$
where  $V_i = \{1 \text{ if involved } 0 \text{ if not } 1\}$ 

and j ∈ J, the power set over I i.e. the set of all subsets of I

R. described by V. is called an <u>impact set</u>. Once the impact set induced by a next change is known, all other modules drop out of consideration. During the <u>identification phase</u> effort is spent on selecting the elements of the impact set from the candidate set i.e. the power set. The complexity of this effort is then simply the a priori uncertainty or the entropy, which is removed by the identification of the impact set:

$$H(V) = -\mathbb{Z}P(V_i)(\log_2 P(V_i))$$

where  $P(V_j)$  is the probability of occurrence of  $R_j$ .

H, the usual formalization of uncertainty, can be interpreted, for example, as the minimum average number of binary decisions necessary to find the impact set R. This measure suits well our intuition and appears proportional to the effort needed for identification. If R. is known a priori, then  $P(V_{\cdot})=1$  and all other impact sets have zero probability. Consequently H(V)=0 and there is no identification necessary.

The maximum effort coincides with maximum entropy. This happens when all impact sets are equally likely and

$$P(V_j) = \frac{1}{|J|} = 2^{-n} + hus + max = n$$

Furthermore, the uncertainty decreases with the number of modules or with any departure of the probabilities from the uniform distribution. After all, identification <u>is</u> easier with a non-uniform distribution.

Once the impact set modules are collected, changes within this set must be <u>coordinate</u>. Effort C of this <u>phase</u> is a bit more difficult to model although it seems obvious that here too the measure should increase monotonically with the size n.of impact set R. However, more research is needed in this area and we can offer here only a few ideas for alternatives:

$$C \sim \frac{n_i(n_i+1)}{2} \sim n_i^2$$

based on the assumption that changes in modules must be coordinated pairwise

where m is the number of choices to be made for each

namely the number of possible design sequences through the set of modules

Perhaps the easiest to capture is the implementation phase.

The impact set is already identified, he change designed, i.e. its individual components coordinated. All that remains is to execute and document it. (Complications due to several change teams working concurrently are not considered in this paper.)

The most plausible way to characterize this simplest of the three phases is to assume that work W spent is proprotional to the impact set size

So far our analysis of the three phases of software modification activity starts with the next change assumed to be given, while the identity of the impact set is considered uncertain. Now we introduce the notation of the <a href="https://doi.org/10.1007/journal.org/">https://doi.org/10.1007/journal.org/</a>

$$Q_k = \{s_i; i \in I_k\}$$
 keJ

which is a collection of modules as explicitly presented at next change time: the resulting impact set must then be identified as described earlier.

The hit set  $Q_k$  can also be described by the vector

$$U_{k} = (v_{1}, v_{2}, ..., v_{i}, ..., v_{n})_{k} \quad k \in J$$
(the power set)
and
$$v_{i} = \begin{cases} 1 & \text{if } i^{\text{th}} \text{ module is in hit set} \\ 0 & \text{if not} \end{cases}$$

 $P(U_k)$  is the probability of the collection  $\mathbf{Q}_k$  hitting the system.

The total modification uncertainty over the product set (or matrix) UxV can be derived from their joint distribution. Using conditional probabilities

thus decomposing the entropy into two terms. The first one represents the uncertainty of the hit - characterizing the modification process; while the second term stands for the additional uncertainty about the actually impacted modules-a pure system property.

We would also like to emphasize the fundamental difference between identification and coordination efforts. They contribute in quite different ways to the difficulties of software system modification. Maximum coordination, for example, is needed with the largest impact set which may be coupled with minimum identification entropy: only one impact set is of size n. On the other hand minimum (zero) coordination occurs in impact sets of unit size and only impacts sets greater than one have probability zero, which still leaves the identification effort to vary up to  $\log_2(n+1)$ .

#### CONCLUDING COMMENTS

A good application of the ideas presented in this paper could be to systematically test them against the results of behavioral studies <WEI>. We also believe in the relevance

of some basic concepts of information theory to software complexity. For instance, information can be defined only in the context of uncertainty (namely, for an answer you must first have a question). Therefore the top down method of design may be an oversimplification: iterative steps between exploring the problem (generating uncertainty) and selecting a particular solution to it (reducing uncertainty) appear to be closer to real life. But more concrete and immediately applicable guidelines can be drawn from our approach:

- Design for small impact sets try to localize potential, though a priori not completely known, modifications
- Modularize systems by the relative frequency of expected modifications
- Organize documentation by providing the fastest access to areas which most likely change
- Monitor hit sets and impact sets during maintenance for planning

- <AHO> Aho, Hopcroft, Ullmann, "The Design and Analysis of Computer Algorithms," Addison Wesley 1975.
- (BAS) V. R. Basili et al, "The Software Engineering Laboratory," Technical Report TR-535 SEL-1 University of Maryland, May 1977
- <BEB> H. Beilner and L. A. Belady, "Speculations on Program Complexity," Unpublished paper, IBM Research 1975
- <BEL> L. A. Belady and M. M. Lehman, "A Model of Large Program Development," IBM Systems Journal, Vol. 15, No. 3, 1976.
- <CHA> G. J. Chaitin, "Information Theoretic Computational Complexity," IEEE Transactions on Information Theory, IT-20, 10-15, 1974.
- <CON> J. P. Considine, "A Computable Measure of
   Fragmentation for Direct Access Volumes," IBM Research
   Report RC 6241, Oct. 1976
- (EDW) N. P. Edwards, "On the Reduction of Complexity and the Verification of Software," IBM Research Report, Yorktown 1975 (available from author).
- (ELS) J. L. Elshoff, "An Analysis of Some Commercial PL/I Programs," IEEE Transactions of Software Engineering, Vol. 2, No. 2, 1976.
- (GIL) T. Gilb, "Software Metrics," Winthrop 1977
- <HAK> S. L. Hantler and J. C. King, "An Introduction to Proving Correctness of Programs," Computing Surveys, Vol. 8, No. 3, Sept. 1976
- <HAL> M. Halstead, "Elements of Software Science".
- <HAN> F. M. Haney, "Module Connection Analysis-A Tool for Scheduling Software Debugging Activities," Fall Joint Computer Conference 1972.
- (HEL) L. Hellerman, "A Measure of Computational Work," IEEE Transactions on Computers, Vol. 21, No. 5, 1972.

- (HIM) D. M. Himmelblau, "Decomposition of Large Scale Problems," North Holland, 1972.
- (HOC) D. H. Hooton, "A Case Study of Evolution Dynamics," M. Sc. Thesis, CCD, Imperial College, London, Sept. 1975.
- (JON) Ch. J. Jones, "Design Methods," Wiley Interscience 1974.
- KNU> D. E.Knuth, "An Empirical Study of FORTRAN Programs," Software Practice and Experience Vol. 1, 1971.
- <MCC> T. J. McCabe, "A Complexity Measure," IEEE Transactions on Software Engineering Vol. 2, No. 4, Dec. 1976.
- (MIL) H. D. Mills, "The Complexity of Programs (in Program Test Methods)," Prentice Hall, 1973.
- <MYE> G. J. Myers, "Reliable Software Trough Composite Design," Petrocelli-Charter, 1975.
- (SIC) Special Interest Committee on Software Engineering, ACM, "Language Design for Reliable Software," as summarized in Software Engineering Notes, (Workshop in Raleigh, N. C., March 1977).
- (SIM) H. A. Simon, "The Sciences of the Artificial," MIT Press 1969.
- (SYM) L. R. Symes and R. R. Oldehoeft, "Context of Problem Solving Systems," IEEE Transactions of Software Engineering, Vol. 3, No. 4, July 1977.
- (VAN) M. H. van Emden, "An Analysis of Complexity," Mathematisch Centrum, Amsterdam, 1971.
- <WAL> C. E. Walston and C. P. Felix, "A Method of Programming Measurement," IBM Systems Journal No. 1, 1977.
- WEA> W. Weaver, "Science and Complexity," American Scientist, Vol. 36, p.536, 1948.
- (WEI) L. Weissman, "Psychological Complexity of Computer Programs," SIGPLAN Notices 9, June 1974.
- <WHY> Whyte-Wilson-Wilson, "Hierarchical Structures," American Elsevier, 1969.

<WOL> R. W. Wolverton, "The Cost of Developing Large-Scale Software," IEEE Transactions on Computers, Vol.23, No. 6, 1974.

COMPUTER SCIENCES CORP ARLINGTON VA

SOFTWARE PHENOMENOLOGY - WORKING PAPERS OF THE SOFTWARE LIFE CY-ETC(U)

AUG 77 B ELKINS, L HUNT

DAHC26-76-D-1006 AD-A053 014 UNCLASSIFIED NL 5 OF **8** O53 014

## Potential Impacts of Software Science

on

Software Life Cycle Management

M. H. Halstead
Purdue University
CSD-TR 237
July 1977

### Prepared for:

Human Factors, Individual and Group Productivity Measures Session, Software Life Cycle Management Workshop U.S. Army Computer Systems Command, Airlie House, Airlie, Virginia, August 22-23, 1977

Research sponsored in part by NSF Grant No. MCS 7605611

# Potential Impacts of Software Science on Software Life Cycle Management

M. H. Halstead Purdue University

## Abstract

A listing of present needs in Software Engineering is followed by a brief discussion of new and "quasi-complete" engineering disciplines and their relation to corresponding branches of natural science.

Recent results in Software Science are then described, suggesting a natural science base. The paper concludes with specific suggestions of areas in which these results might well provide guidance and insight into problems of software development and maintenance.

## Engineering Needs

The successful management of large programming projects over their complete life cycles depends largely upon the discipline of Software Engineering. But in its present state of development this discipline still falls far short of being a "quasi-complete" branch of engineering.

Substantial progress is urgently needed on many fronts. A handful might be mentioned. 1). Problem Specifications. Software Engineering should be able to provide optimal methods for obtaining them, resolving their ambiguities and incompatibilities, determining their completeness, and the "best" way to present them to programmers. 2). Programmer Productivity. In this area, a "quasi-complete" discipline would provide guidelines for programmer selection, training, and subsequent evaluation; as well as quantitative methods for estimating the man-hours and organization required to achieve any well specified goal. 3). Program Testing. Quantitative methods are needed for estimating the effects of source language, modularity, program level or complexity, volume and program clarity upon program testing and maintenance. 4). Job Scheduling. More reliable techniques are needed for estimating man-power requirements of a project as a function of specifications, programming language, team size and mix, memory constraints and required reliability. 5). Optimizing Implementation versus Maintenance costs. A quasi-complete engineering discipline should provide quantitative guidelines for sound tradeoff studies in this important but complex area.

## Natural Science and Engineering

For any engineering discipline to be quasi-complete, it must rest upon and be grounded in a "hard" natural science, a science with sound metrics,

reproducible experiments, and dependable "laws". In virtually all cases, branches of engineering have preceded (and perhaps stimulated) the natural science upon which they are now based. During that time, however, their value to mankind was severally limited. But now, for example, aeronautical engineering based on fluid dynamics, power engineering based on thermodynamics, electrical engineering based on electrodynamics, and mechanical engineering based on statics, dynamics and strength of materials may all be considered quasi-complete, highly competent, and useful.

## Software Science

A considerable body of evidence now exists which suggests that the metrics, methods, and hypotheses of software science [7] may be capable of providing such a base for software engineering. It is pertinent to note, however, that theories are not theorems. They require independent experimental confirmation at more than one laboratory. Unlike mere mathematical models, theories must provide new insight into natural phenomena, and they only become important when they are shown to predict previously unrecognized and unexpected relationships in areas beyond their originally intended scope. Further, a theory is never complete, but continues to be used only until its recognized inadequacies can be eliminated by a new theory.

Software science is based on a handful of language independent parameters which can be measured (or counted) directly from any hard copy or computer program. These are the number of unique operators  $(n_1)$ , the number of unique operators  $(n_1)$ , and

the total usage of operands  $(N_2)$ . A fifth parameter, the number of conceptually unique input/output operands  $(n_2^*)$  required by a procedure call upon the program is also an important language-independent metric.

These basic metrics are not independent, and a number of quite useful relationships have been found among them. For example, denoting the sum of  $N_1$  and  $N_2$  as the length  $N_2$ , it has been found that programs tend to obey the relation

$$N = n_1 \log_2 n_1 + n_2 \log_2 n_2$$

Further, denoting the sum of  $\eta_1$  and  $\eta_2$  as the vocabulary  $\eta_1$  the program volume V is

The potential (or least possible) volume V\* is

$$V* = (2 + \eta_2^*) \log_2 (2 + \eta_2^*)$$

which gives an implementation level L of

$$L = V*/V = \frac{2}{n_2} \frac{n_2}{N_2}$$

It follows that the product

is invariant under translation from one language to another.

It has also been found that the number of elementary mental discriminations (E) required to produce a program should be given by

This leads directly to an estimate of programming time (T). Using the Stroud Number (S) or 18 elementary mental discriminations per second, gives

Initially unforseen relationships derivable from the basic metrics include Ostapko's [10] derivation of Rent's Rule for circuit to pin ratios in hardware, Elci's [3] demonstration that the lengths of operating systems are functionally related to the number of their allocatable resources, and Funami's [5] demonstration that programming error rates are related to E. Perhaps the most interesting unexpected finding to date is the observation that the relationships governing computer programs can be applied to technical prose as well.

With respect to deeper understanding or insight, one might list the areas of program purity or "impurity classes", the role of modularity, the quantitative effects of "GO-TO's", the measurement of clarity, and most recently some apparent insight into the learning process itself.

Independent experimental verifications of various facets of the overall theory have been published by Bohrer [2] of Illinois, Elshof [4] of General Motors, Bell and Sullivan [1] of Mitre, Ostapko [10] of IBM and Love and Bowman [9] of General Electric.

# Experimental Methodology

To illustrate the relationships and methodology discussed above, we will first present the results and analysis of a simple experiment, and follow with a few summaries of previously published data.

In January 1977 a class of 28 advanced graduate students at Purdue individually programmed Euclid's greatest common divisor algorithm in Fortran, and counted the software parameters in their own versions. Their results are given in Table 1. Students 10, 16 and 27 neglected the implied "End-of-line" operator in Fortran, so their reported values of  $n_1$  have been increased by one, and their values of  $N_1$  have been increased by 12.

Table 1. Software Parameters of 28 Independent Fortran Versions of the GCD Algorithm

Student	n <sub>1</sub>	n <sub>2</sub>	N <sub>1</sub>	N <sub>2</sub>	Student	n <sub>1</sub>	n <sub>2</sub>	N <sub>1</sub>	N <sub>2</sub>
1	11	6	34	21	15	11	6	34	21
2	11	5	32	19	16	12	7	34	21
3	12	6	34	21	17	12	6	38	21
4	12	6	34	21	18	11	6	33	21
5	10	6	31	21	19	12	6	34	21
6	12	6	34	21	20	11	6	34	21
7	11	6	34	21	21	10	6	34	21
8	11	6	31	21	22	11	6	34	21
9	12	6	34	21	23	11	6	32	21
10	12	6	36	21	24	12	6	35	21
11	11	6	35	21	25	12	5	33	19
12	11	6	34	21	26	12	6	32	21
13	11	5	33	19	27	11	6	34	21
14	12	6	34	21	28	1.0	6	31	21
					MEANS	11.32	5.93	33.64	20.79
					5.D.	.67	.38	1.50	.63

Using the individual values in Table 1, a number of the software relationships can be evaluated, and the degree of conformity calculated.

<u>Length</u>

Obtaining the observed value of length (N) from

and the estimated length (A) from

$$\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$$

gives mean values of

$$N = 54.43 \pm 1.75$$

$$\hat{N} = 54.90 \pm 3.76$$

$$(N-\hat{N})/N = -0.009 + 0.058$$

# Implementation Level

Obtaining the observed potential volume (V\*) from the condition that a procedure call on a GCD algorithm must have two inputs and one output, or  $n_2^* = 3$  in

$$V* = (2 + \eta_2^*) \log_2 (2 + \eta_2^*)$$

and the observed volume (V) from

$$V = (N_1 + N_2) \log_2 (\eta_1 + \eta_2)$$

allows the observed level (L) to be calculated from

The estimated level (L) is obtained from

$$\hat{L} = \frac{2}{n_1} \frac{n_2}{N_2}$$

Then for Table 1, the mean values are

$$L = 0.0529 \pm 0.0023$$

$$\hat{L} = 0.0505 \pm 0.0035$$

$$(L-\hat{L})/L = 0.028 \pm 0.067$$

# Potential Volume

Obtaining the estimated potential volume  $(\hat{V}^*)$  from

$$\hat{V}^* = \hat{L}V = \frac{2}{n_1} \frac{n_2}{N_2} (N_1 + N_2) \log_2 (n_1 + n_2)$$

H

n

and the actual potential volume (V\*) from  $n_2$ \* = 3 yields

$$(V*-\hat{V}*)/V* = 0.029 \pm 0.067$$

# Vocabulary

Obtaining the observed vocabulary (n) from

$$\eta = \eta_1 + \eta_2$$

and the estimated vocabulary  $(\hat{n})$  by solving iteratively for n as a function of N  $\forall n$ 

$$N_1 + N_2 = n \log_2 (n/2)$$

gives the mean values

$$\hat{n}(N) = 17.42 \pm 0.38$$

$$(n-\hat{n}(N))/\eta = -0.016 \pm 0.038$$

# Unique Operators

Using the vocabulary  $\hat{n}(N)$  estimated from length as calculated above, the estimated unique operator count  $\hat{n}_1(N)$  can be obtained from

$$\eta_1 = (n-B)/(A+1)$$

where

$$A = \frac{n_2 * \log_2(n_2 * / 2)}{2 + n_2 *} ; B = n_2 * - 2A$$

The data in Table 1 give the following average values

$$n_1 = 17.31 \pm 0.67$$

$$\hat{n}_1(N) = 11.20 \pm 0.28$$

$$(\eta_1 - \hat{\eta}_1(N))/\eta_1 = 0.008 \pm 0.055$$

# Unique Operands

In the same way, the observed  $\,\eta_{2}^{}\,$  can be estimated from  $\,\eta_{2}^{}\star\,$  and length via

$$\hat{n}_2(N) = (\hat{An}(N) + B)/(A + 1)$$

And again, the Table 1 data show

$$\eta_2 = 5.93 \pm 0.38$$

$$\hat{\eta}_2(N) = 6.23 \pm 0.10$$

$$(\eta_2 - \hat{\eta}_2(N))/\eta_2 = -0.054 \pm 0.064$$

Summarizing the relative errors, we have

Length (N)	-0.009 ± 0.058
Level (L)	0.028 ± 0.067
Potential Volume (V*)	0.028 ± 0.067
Vocabulary (n)	-0.016 ± 0.038
Operators (n,)	0.008 ± 0.055
Operands (n <sub>2</sub> )	-0.054 <u>+</u> 0.064

This can be taken as evidence that for a very small program, replicated by 28 highly fluent programmers, the software hypotheses tested gave reasonably good agreement with the observations.

In order to illustrate the applicability of these relationships to programs large enough to be of practical interest, Elshof's [4] data validating the length equation for commercial PL/I programs are given in Table 2.

Table 2. Elshof's PL/1 Data

	Number of Programs in Size Class	Length Observed (N)	Length Predicted (N)
	3	18,592	19,091
	17	10,685	11,049
	23	5,751	6,005
	39	3,165	3,318
	17	1,590	1,663
	9 11 Sam 23 29	831	911
	4	369	522
	5	198	195
	1	122	129
Totals	120	41,303	42,883

FIGURE 1. ELEMENTARY MENTAL DISCRIMINATIONS

- O GORDON-HALSTEAD DATA
- X JOHNSON DATA
- WALSTON-FELIX DATA

SOLID LINE EXTENDS FROM 5 MINUTES TO 1000 MAN-YEARS, 5=18 disc./SEC.

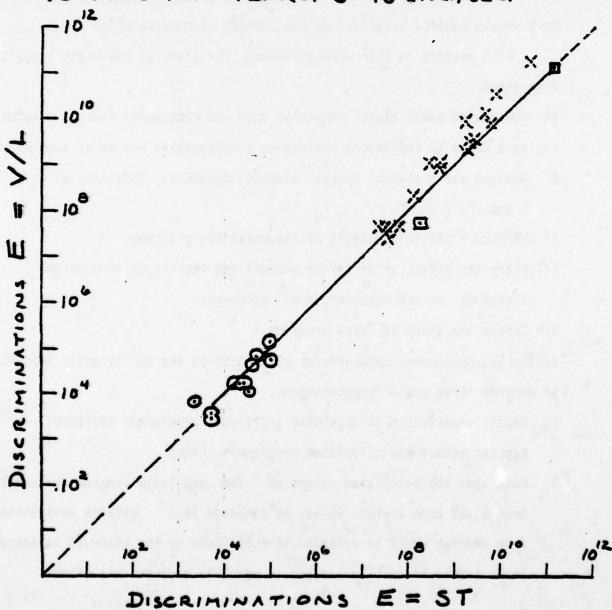


Figure 1 has been taken from another paper [8], which indicates that projects ranging from well under one day to well over one hundred man years follow the Effort Equation in a general way.

Indicated Studies

A number of areas in which software science might prove useful to Software Life Cycle Management come immediately to mind. While each of the five areas cited in the introduction as needing substantial improvement should benefit directly, we can perhaps be more specific.

With respect to task specifications, for example, one might suggest five steps.

- 1) Start with small tasks, requiring only one programmer from 10 minutes to 8 hours to implement, and gather a substantial number of samples.
- Analyze the technical English problem statements, obtaining n, N, V, L and V\*, E and T.
- 3) Perform a similar analysis of the resulting programs.
- 4) Study the effect of different methods and techniques of problem statement on the resultant small programs.
- 5) Expand the study to large programs.

  Similarly, programmer productivity relationships are sufficiently important to warrant large scale investigations.
- Repeat experiments to determine individual programmer variances between actual and calculated programming times.
- Note that the calculated values of T for very large programs have all been based upon average values of language level. Because deviations from average would be expected to contribute to the observed variance in T, it should be illuminating to actually measure this effect.

- 3) For a significantly large data base, obtain the statistical variance between observed and calculated programming times.
- 4) Perform quantitative studies on the effect of E of existing programs on the rate of error discovery by new programmers. This should yield a measure of their fluency (and concentration).
- 5) Investigate the possibility that programming aptitude might be estimated by a software analysis of a technical prose paragraph written by a candidate for programmer training.

The area of program testing might benefit by further investigations of the software relationships. This might require

- 1) Sharpening the definition of "Delivered" bugs.
- Development of a large data base, with samples from most of the widely used languages.
- Repetition of experiments to determine the expected variance between observed and calculated error rates.
- 4) Analysis of modularity, following Zislis' [11] use of software science for program testing.
- 5) Use of software error rate relations in predicting remaining errors as a function of expected errors and errors removed.

The area of job scheduling is related to that of programmer productivity, but requires other information as well. Consequently, it involves a number of additional points.

 Because any two independent software parameters determine all others, it follows that the task specifications and the language to be used determine, in principle, the time to be required. In practice, however, it is not that simple. For example, even a concentrating programmer, fluent in the language, without computer memory constraints, must start with a complete problem statement. Furthermore, a problem statement which contains no contradictions or ambiguities may be "complete" for one programmer, but not for another. Nevertheless, the existence of a basic relationship between the number of conceptually unique input/output operands  $(n_2^*)$ , the language level  $(\lambda)$ , and the time (T) suggests that an intensive investigation is now possible and warranted.

2) It has been observed that the product LV = V\* is invariant when a given algorithm is translated from one language to another. But within any one language, L decreases as the potential volume increases. Consequently, a given language can be characterized by its language level  $(\lambda)$ , defined as

 $\lambda = LV*$ 

Algebraically, this results in

 $T = V/SL = V*3/S\lambda^2$ 

and consequently  $\lambda$  is a parameter of considerable interest. While the mean value of  $\lambda$  appears to lie somewhere near one for a number of programming languages, it has a large variance which appears to increase as the mean increases. Because the data so far available is based on small samples of small programs, it can not be used with confidence. Therefore, statistical determinations of means and variances of  $\lambda$  for any language of interest should be made. This study might well include investigations of the effect of different programming methodologies within a single language. It could also

be extended to an analysis of any proposed new language. It could then serve in trade-off studies on cost versus benefits of change to a higher level language, or the question of special purpose versus general purpose languages.

With respect to the problems of optimizing implementation costs versus maintenance costs, it appears quite likely that the quantitative approach provided by software science can be of considerable value. This results from recent work of Gordon [6], who has shown that the measure of elementary discriminations E is in an interesting sense ambiguous.

In the usual case of program implementation, E does indeed measure the time required to develop the program. If, however, additional time is then spent in improving or increasing the legibility of the program, the effect is to reduce the final value of E, rather than to increase it. The final value of E for an improved program then represents not the total effort to write it, but a measure of the effort to understand it — a measure of clarity.

This suggests that a quantitative measure of clarity could be made before a program is polished. Then, the amount of effort which could advantageously be used in increasing the clarity could be determined on the basis of the needs of maintenance.

### References:

- [1] Bell, D. E., and J. E. Sullivan. "Further Investigations into the Complexity of Software". MITRE Technical Report 2874, Vol. II, June 30, 1974.
- [2] Bohrer, Robert. "Halstead's Criterion and Statistical Algorithms." Proceedings of the Eighth Annual Computer Science/Statistics Interface Symposium, Los Angeles, February 1975, pp. 262-266.
- [3] Elci, Atilla. "Factors Effecting the Program Size of Control Functions of Operating Systems." Ph.D. Thesis, Purdue University, December 1975.
- [4] Elshoff, James L. "Measuring Commercial PL/1 Programs Using Halstead's Criteria", ACM SIGPLAN Notices Vol. 7, No. 5, May 1976.
- [5] Funami, Yasuo, and M. H. Halstead. "Software Physics Analysis of Akiyama's Debugging Data", Proc. MRI XXIV International Symposium: Software Engineering. New York. Polytechnic Press, 1976.
- [6] Gordon, R. D., "A Measure of Mental Effort Related to Program Clarity", Ph.D. Thesis, Purdue University, August 1977.
- [7] Halstead, M. H. "Elements of Software Science", Elsevier North-Holland Publishers. New York 1977.
- [8] Halstead, M. H. "A Quantitative Connection Between Computer Programs and Technical Prose," Digest of Papers from COMPCON 77 Fall. IEEE Computer Society.
- [9] Love, L. T. and A. B. Bowman, "An Independent Test of the Theory of Software Physics". ACM SIGPLAN Notices, Vol 11. No. 11, November 1976, pp 42-49.
- [10] Ostapko, Daniel L. "On Deriving a Relation Between Circuits and Input/Output by Analyzing an Equivalent Program". ACM SIGPLAN Notices, Vol 8, No. 6, June 1974, pp 18-24.
- [11] Zislis, Paul M. "Semantic Decomposition of Computer Programs: An Aid to Program Testing", ACTA Informatica, Vol 4, 1975 pp 245-269.

# NOTES ON SOFTWARE RELIABILITY AND QUALITY

R. C. McHenry August 1977

International Business Machines Corporation Federal Systems Division Gaithersburg, Maryland 20760 Notes on Software Reliability and Quality

#### 1. Introduction

The notes are intended to review the subjects of software reliability and quality from a managerial perspective. As such the notes are intended to round out the coverage of the topics provided by experts at the USACSC workshop.

The managerial perspective is the author's: the focus is on projects of substantial scale where formality of organization and approach are required. More than one manager will be involved and their principle tasks are:

- o balancing product, resources, and schedule
- o establishing and maintaining a process which affords the visibility to appraise and redirect progress.

The second principle task orients the author's research in software process engineering.

The notes are organized into sections which comment on software reliability, system perspective, quality assurance, a quality index, and data. The software reliability note is adapted from a 1975 note by a managerial colleague, Donald O'Neill. If the software reliability note appears inconclusive, then it has succeeded, since no approach to software reliability appears to be preeminant. The brief system perspective note suggests an alternative approach to software reliability: addressing the larger question of system reliability. The quality assurance note criticizes the formal quality assurance programs required for U.S. weapon system development. The quality index note examines a proposed index.

The final note issues an appeal for collecting and publishing data to support software reliability and quality research.

## Software Reliability

Reliable software is software that does not fail. However, large and complex software systems sometimes fail. The question is not whether the software is reliable, but rather how often it can be expected to fail.

The temptation to define software reliability in the well understood framework of hardware reliability must be resisted until the salient differences between hardware and software are examined. All software errors contribute to software reliability. Given the same data and conditions software operations (errors) are repeatable. The difficulty in recreating the same conditions (data and timing) is such that a problem log must retain all unresolved problems so that multiple occurrences of an infrequently appearing software error can be correlated to diagnose and correct the defect.

At initial operation, hardware reliability may exceed software reliability. However, as software faults are corrected, the software reliability should exceed hardware reliability.

Certain boundaries must be placed on the conditions under which software reliability is appraised. In order for a system error to be charged to software, it must occur when all hardware is operating correctly. For example, the execution of an illegal instruction due to a hardware transient resulting in a subsequent propagation of errors through software should not be attributable to software. To some degree, software can compensate for transient hardware errors by defensive coding, (e.g., retrying a failed element). In addition, software reliability measures should be limited to testing or using software within the limits and

constraints of a design envelope. If these limits are exceeded, the software should not be charged for performance related errors resulting from the out-of-specification conditions.

In developing software many instructions are written. There is potential for introducing error in each instruction, (e.g., compares and branch instructions used in implementing flow of control are most error prone; followed by instructions used for data operations). When the code is executed, some segments are executed often, others infrequently. Through use the software most executed becomes most reliable. It takes longer to uncover problems in seldom used segments.

Software reliability can be approached from two directions: past success and future failure. By evaluating past success, the proportion of the system that has been successfully demonstrated is determined. Software, unlike hardware, can be depended upon all the time to perform correctly in operational use for those areas demonstrated. The test program provides the means for establishing proven capability. Since software is repeatable, a test program that successfully exercises the operational scenario certifies the software reliability. Although every system instruction can be tested, it is not possible to test every combination of data and timing variations. However, the concept that reliability can be calibrated through the relationship of test to operation has real value.

A test coverage matrix can help evaluate effectiveness. The test program requires, in effect, a test of the test program. A technique for testing the test procedures is comprehensive code reading following each test phase. The effectiveness of the test program can be judged by the number of additional errors detected.

The approach to quantifying failures uses past failures to project future failures. The key is to record and categorize the errors detected in each phase of development. Since the number of errors encountered is a function of the number of errors still in the software, then this record predicts future failures.

A simple error removal model is suggested: some fraction of the remaining error population is identified and corrected by each distinct software testing activity. Each distinct activity must add a new dimension rather than provide more-of-the-same testing. This simple model uses two factors: the error removal rate and the number of testing steps. The error removal model confirms that top-down approaches should increase system reliability. If there are N test periods, the first increment (system control) is tested N times, the second increment is tested N-1 times, with the last increment being tested perhaps only once.

For any system there is a reliability figure independent of whether it is known. The notion of dependability requires that the software reliability figure be known, and the real significance of determining or predicting the software reliability lies in the use of error information. If decisions need to be made based on the level of software quality assurance, then consideration should be given to establishing permissible software error budgets that must be achieved. It may, in fact, be more effective to manage the error budget than attempt to predict the ultimate reliability. The penalty for low reliability is operational impact and high maintenance cost. However, the cost and schedule impact resulting from excessively high software reliability emphasis may be unnecessary. By establishing the required objective and managing to it, the resulting software satisfies the operational need within the cost and schedule boundaries.

Another consideration is the effect of the software error on the operation of the system. A catastrophic error (due to a computer stop, software interlock, memory destruction, or input/output difficulty) results in the inability of software to continue processing. A second level of software error is characterized by the continuation of the processing following the error but with an unknown effect on the results. The effect of a software error may be predictable and minimal. An error severity category may be useful in terms of determining where the test emphasis should be. A comprehensive test program should remove all catastrophic errors.

Error prone segments account for a large proportion of the life cycle errors. For example, Jones (1) reports about 4% of the modules account for perhaps 38% of all OS/360 defects. Traditionally, error prone segments have had complex flow of control (simplified now through structured programming) or complex data interfaces to other segments. Identifying error prone segments early in the process substantially improves reliability through remedial action and test emphasis.

#### 3. System Perspective

We might attack software reliability by determining measures for system reliability. At present, system reliability calculations take only hardware into account. The inherent assumption is perfect software reliability. As a result there is no margin for software error since the reliability budget usually has been consumed with expected hardware failures. While the hardware errors may be transient, intermittent, or solid, only solid errors form the basis for the reliability calculation.

A simple example illustrates the possibility of a system perspective. Devices satisfying a specification of one bit lost in  $10^9$  bit transfers were attached to a computer which demanded  $10^6$  bits per second without

parity checking. System failures which occurred approximately every 15 minutes were attributed to software. A recovery procedure implemented in software eliminated system failures from this source. We should resist the urge to place all the blame on the designers because the components met their individual solid failure budgets and the designers had little help from traditional hardware reliability models. The software as originally implemented also met its original specification. Furthermore, while the example has been presented to highlight the problem source, the "intermittent" problem was difficult to diagnose from the evidence in memory dumps which were not always taken because of the pressures attendent to system integration.

Another example may help illuminate the system perspective. In the design of a high availability system, two computers were specified: one active, the other a "hot" standby. Following a switchover, the design called for the "failed" computer to be tried, without repair, as the standby. The rationale was simply intuition concerning the relative frequency of non-solid and solid hardware faults. The intuition appears justified in operation where recoveries, including those required by the standby system, exceed repairs.

The system perspective suggests several questions concerning hardware, software, and system reliability measures. Some of these questions warrant discussion during the workshop:

- o how much emphasis should system reliability be given?
- o can system reliability considerations guide software reliability research?
- o should we perceive software as the system and somehow subordinate hardware?

# 4. Quality Assurance (2)

One quality assurance (QA) definition is the formal approach to systematically check the development of a product to assure the product conforms to requirements. In this definitional context, quality assurance can be paraphrased as conformity assurance or adequacy assurance.

One military specification (3) identifies several software QA program areas:

- o work tasking and authorization procedures
- o Configuration management and library control
- o software documentation
- o reviews and audits
- o testing
- o corrective action
- o tools, techniques, methodologies

Distinction between QA and testing is attempted in some contractual terms. The distinction is usually organizational with QA given an autonomous posture which frees QA from the demands of delivery schedules. The distinction is difficult below the grossest level of detail. The figure attempts to distinguish quality reviews in a project context. The figure identifies two types of quality reviews: one associated with design reviews and the other associated with testing.

Since design reviews are a substitute for objective execution tests, the associated quality review requires specific focus to avoid duplicative effort. The QA focus must establish, first, a role as requirements advocate to assure the requirements are not lost in assessing technical and implementational feasibility of the system design; and, second, a role as test reviewer to assure the adequacy of the test design.

The quality review associated with testing is more mechanical than the design associated review, because QA is more nearly a witness than a participant role. The quality review assures the test environment (hardware, software procedures, data) and test results.

One of the central QA problems is the definition of software quality. Two schools of thought concerning software quality are:

- o Quality is measured by the number of software errors reported by users.
- o Quality is measured by the length of time software runs without failing (stopping).

The first has the difficulty of either having no error budget (implicitly zero defects) and having only negative evaluation as errors are detected; or having an error budget and having to admit frailty in advance. The second has the difficulties associated with software reliability discussed previously. In spite of these difficulties some guidance is derived; the QA activity must include:

- o Procedures for reporting detected errors
- o Closed loop means to ensure error correction

An often overlooked QA problem is accommodating customer, contractor, or computer manufacturer furnished software.

In addition to the software itself, QA cognizance includes documentation. In some projects, QA cognizance is limited to documentation and in degenerate cases is an editing function.

The software development process itself must implement the QA discipline. This maxim should be discussed during the workshop to determine its value in appraising or disciplining research.

The implementational difficulties also arise in distinguishing user perceived functions versus components used in implementing function and in computing availability. By assuming catastrophic errors have been removed prior to operational use, a gross approximation of A may be made based on system availability. Under this assumption an essentially available system quality would be influenced only by corrective maintenance:

$$Q = \frac{1}{1+M}$$

No difficulty is forseen in establishing cost accounts to distinguish corrective activity from enhancement or other supportive activities.

Substituting for A and M (equations 1-3) we obtain

(4) 
$$Q = \frac{\Sigma f_i a_i}{1 + \frac{C}{B}}$$

This form shows flaws in the construction. For a software system of a given availability, as the maintainer, we can maximize Q by doing nothing, i.e., C=0 or by having a total support budget B >> C, e.g., get as much enhancement budget as possible and keep C marginal. Further, in the interest of maximizing Q, we would not fix a problem to restore  $f_i$  if

$$f_i < \frac{\text{cost to fix } f_i}{B}$$

since to do so would increase denominator more than numerator in (4). This could be avoided if the "cost to the user" of not having  $f_i$  repaired were reflected in some recurring cummulative fashion - perhaps  $f_i t_i$  where  $t_i$  is time left unrepaired.

# 5. A Quality Index

This note explores the potential use of functional availability and corrective maintenance cost as quality measures. The intent is to stimulate rather than to recommend the specific quality measure which is presented.

# 5.1 Operational Quality

Cave (4) has suggested a measure of operational software quality based on functional availability A and maintenance cost M:

$$(1) \quad Q = \frac{A}{1+M}$$

where (2) 
$$A = \sum_{i} f_{i} a_{i}$$

$$f_{i} = \text{importance of ith function}$$

$$1 = \sum_{i} f_{i}$$

$$a_{i} = \text{availability of ith function}$$

$$1 = \sum_{i} a_{i}$$

and (3) 
$$M = \frac{C}{B}$$

$$C = cost of corrective maintenance$$

$$B = total support budget$$

The suggested measure is appealing because of its conceptual simplicity and the general interest in the measures of A and M. Availability is of critical interest to the user and to the developer and its component functional importance is needed by the designer of the various reduced function modes in which a complex system may operate. The ratio of corrective maintenance to total support budget is of interest to both managers and researchers.

The suggested measure has difficulties in compariative use and in implementation. Two systems which have identical quality indices can not be compared. For example:

$$Q_1 = Q_2 = 0.45$$
, however  
 $Q_1 = \frac{0.45}{1+0}$   
 $Q_2 = \frac{0.9}{1+1}$ 

The construction  $\Sigma f_{ia}$  suffers from the implied requirement of a prior definition/identification of all "functions" and their importance. Alternatively, the mere existence of an open (unduplicated) problem report E implies a 'function' denied  $(a_i = 0)$  and the priority assigned to E is a measure of the importance of the function  $(f_i)$  to the user. Latent problems, it can be argued, are quality measures on the software producer but not on the system from the user's viewpoint, i.e., the availability of an unused function is immaterial. Therefore we can indicate quality in some form like

$$Q = f(E)$$

and

A = 
$$\frac{1}{\Sigma E + 1}$$
 or maybe  $\frac{100}{\Sigma E + 1}$ 

or more generally

$$A = \frac{100}{\Sigma p_i E_i + 1}$$
 where  $p_i$  is a normalized E priority

The disadvantage of this particular representation is that A decreases too rapidly with  $\Sigma p_i E_i$ . A better scaling could be provided by  $k \Sigma p_i E_i$ , k = .01.

The implementational difficulties also arise in distinguishing user perceived functions versus components used in implementing function and in computing availability. By assuming catastrophic errors have been removed prior to operational use, a gross approximation of A may be made based on system availability. Under this assumption an essentially available system quality would be influenced only by corrective maintenance:

$$Q = \frac{1}{1+M}$$

No difficulty is forseen in establishing cost accounts to distinguish corrective activity from enhancement or other supportive activities.

Substituting for A and M (equations 1-3) we obtain

$$(4) Q = \frac{\Sigma f_{\mathbf{i}} a_{\mathbf{i}}}{1 + \frac{C}{B}}$$

This form shows flaws in the construction. For a software system of a given availability, as the maintainer, we can maximize Q by doing nothing, i.e., C=0 or by having a total support budget B >> C, e.g., get as much enhancement budget as possible and keep C marginal. Further, in the interest of maximizing Q, we would not fix a problem to restore  $f_{+}$  if

$$f_{i} < \frac{\text{cost to fix } f_{i}}{B}$$

since to do so would increase denominator more than numerator in (4). This could be avoided if the "cost to the user" of not having f<sub>i</sub> repaired were reflected in some recurring cummulative fashion - perhaps f<sub>i</sub>t<sub>i</sub> where t<sub>i</sub> is time left unrepaired.

# 5.2 Development Quality

The quality measure for operating software may have an analog for developing software such that a quality threshhold might be established as an acceptance criterion. The previously described measure can be analyzed in development phase terms to establish a measure for that phase. Consider, for example, substituting the following:

$$A = \frac{\text{successful tests}}{\text{total tests}} \quad \text{and} \quad$$

and given a threshhold of 0.95, 41 tests, one failure, one unit of cost fix, and a 400 unit budget:

$$Q = \frac{\frac{40}{41}}{1 + \frac{1}{400}} = 0.974$$

As in the operating quality case, the difficulty will be in determining A rather than cost accounting M. The difficulty is in determining A during reviews or during debugging under individual programmer control. Since support libraries can count segments and segment updates automatically the following hypothesis is offered:

updates = segments changed as a result of review or test

The hypothesis is dependent on requirement, design, and code segments being small enough to be complete on original entry to the library.

The ratio of C/B is a poor choice since B includes development work and a large development effort changes Q drastically even though the program Q is supposed to measure remains unchanged. Perhaps better would be something like

where  $\Sigma c_i$  is the actual costs of repairs made and C is the cost of the maintenance staff.  $C - \Sigma c_i$  represents the breakage of having to keep required skills even when there are no problems.

 $\frac{\Sigma c_i}{C} \rightarrow 0$  a pretty good program (very few troubles) but high maintenance overhead.

 $\frac{\Sigma c_i}{C} \rightarrow 1$  enough troubles to totally consume budget - little breakage but perhaps buggy program.

Maybe quality really means cost-effectiveness and we are interested in measures like:

- o max {A} for fixed C
- o fixed A for min {C}
- o rate of change  $\frac{dA}{dC}$

The author wishes to express his gratitude to W. C. Cave for suggesting a quality measure for managers which stimulated this note and part of the next. D. L. Walker made substantial contributions to the note.

#### 6. Data

The author has one firm conviction: software quality data must be collected and published to support continuing research. Furthermore, the data must come from real-world software systems rather than small experiments. The key questions are what data and whether that data can be collected from users, developers, maintainers, and operators.

One recommendation of this workshop should be the Army sponsor a working group to identify the reliability and quality data to be collected and collect these data through their in-house and contracted programs.

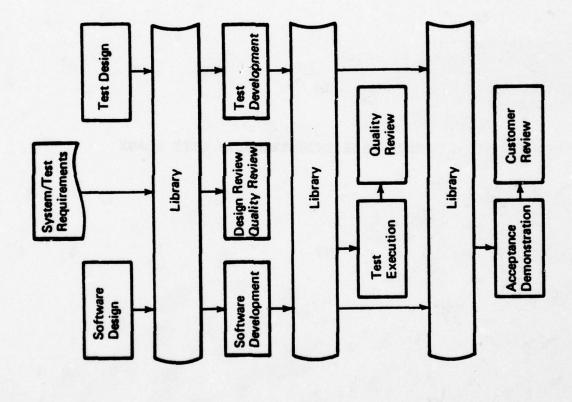
The composition of the working group should include an equal mix of researchers and managers. The mix will yield both direct and indirect results. The possibilities of a researcher-manager working group can be envisioned by creating dialog for the quality measure note. The thought flow in the note is as follows:

- o appeal: simple, interesting components
- o comparative use
- o ambiguity of function/problem priority
- o implementation difficulty
- o cost accounting insight
- o manipulation
- o cost-to-fix insight
- o extensible to other phases
- o implementation difficulty
- o library accounting insight
- o technical dependency
- o cost/budget difficulty
- o cost effectiveness insight
- o other candidates

# References

- Jones, T. C., Program Quality and Programmer Productivity, IBM Corporation, 1976 (presented at Guide 1977).
- 2. McHenry, R. C., and C. L. McGowan <u>The Software Process</u>: <u>Management Engineering</u>, to appear.
- 3. Military Specification, MIL-S-52779, Software Quality Assurance Program Requirements, U.S. Government Printing Office.
- 4. Cave, W. C. and A. Salisbury, Controlling the Software Life Cycle, to appear.

# QUALITY REVIEWS



# A SUMMARY OF SOFTWARE RELIABILITY MODELS AND MEASUREMENT

MARTIN L. SHOOMAN
Department of Electrical Engineering, Division of Computer Science

Polytechnic Institute of New York Brooklyn, New York 11201

With the advent of large sophisticated hardwaresoftware systems developed in the 1960's, the problem of computer system reliability has emerged. The reliability of computer hardware can be modeled in much the same way as other devices using conventional reliability theory; however, computer software errors require a different approach. The paper begins by describing the types and causes of software errors and provides working definitions of software errors and software reliability. Some of the basic data on frequency of occurrence of errors is then discussed. The paper then summarizes and references some of the software reliability models which have been proposed and concentrates on on developed by the author. One of the probabilistic models, (the macro model), predicts reliability based on the initial number of errors in a program, the number removed, and the number remaining in the program. The model constants are calculated from operational test data taken on the software performance. The other, the micro model, focuses on the paths in the program, their frequency and time of traversal, and the error rate along these paths.

# 1.0 INTRODUCTION

#### 1. 1 The Age of Large Computers

The first question that one hears when the term software reliability is mentioned in discussion is. what is that? As the digital computer continues to pervade more and more of our modern technology, we rely on its output more and more for control, data recording, analysis, and decision making. Thus, the size and complexity of the required tasks, the computer hardware, and the computer software has drastically increased in the last three decades. In addition, the costs of software have grown to astronomical proportions. (Recent estimates on the cost of software to the entire U.S. economy range from \$10-\$19 billion.) With such huge size and complexity, it is virtually impossible to definitively specify the problem (without error), to make failure free computer hardware, and to remove all errors from the software.

Along with this growth has come a realization that the largest effort in developing software is due to software integration, test, correction, retest, operational release, correction and rerelease. Actual writing of the first set of code is a small task in comparison. An even more compelling observation is that computers are increasingly being used as the heart of complex real-time systems such as air traffic control, vehicle control, space systems,

and military systems. System reliability is the most important performance measure in such systems.

# 1.2 Some Computer Failure Data

We may shed further light on the distribution between hardware and software errors by considering some actual data. The data given in Table 1-1 lists hardware and software failure rates for a typical real-time data acquisition system. The data represents 9 months of operation totaling 1701 hours. Inspection of the data shows that 48% of the failures were due to software. This is a startling figure if one realizes that although most computer projects estimate and predict the hardware reliability, and use redundancy techniques and high reliability parts to improve the hardware reliability, the software is left to the skill and hard work of the programming team with no quantitative assessment of design progress. We may obtain a typical estimate of what type of system reliability can be obtained in practice. Assuming a simple failure model, we may compute the meantime to failure, MTTF as the reciprocal of the failure rate. Thus, the MTTF = 30 hours for the data in Table I-1. In other words about one failure per day will occur if the equipment is used 24 hours/day and one failure every three days if used 8 hours/day. (Additional data is given in Ref. 2. )

#### 1.3 Hardware vs. Software Failures

In studying the causes of failure, we identify failures by the way they affect the system (mode) and by their causes (mechanisms). In hardware we speak of "early failures" due to poor parts, damage in transportation, and design errors; "middle of life failures" caused by a high load stress exceeding a part's strength; and wear out failures" which occur after much use due to mechanical, electrical, or chemical changes related to age and wear. In the case of software, almost all the errors are design errors. (There may be some errors due to a poor disk, tape, or punched card source of the software or errors in installation.) Also Lehman and Belady in Ref. 3 and 4 describe the compound effect of changes in specifications and added features as Growth Dynamics, which results in an increasing failure rate very much like wear out. Thus, out of the many models used in hardware failure analysis, only the ones describing design errors find wide use in software.

#### 2.0 DEFINITION OF A SOFTWARE ERROR

The following definition of a software error is proposed: One or more software errors exist in a

system if a software change is required to correct system performance so as to meet specifications. Inherent in the above definitions and discussion is the assumption that errors can be and are detected and recorded. The detection of errors can be effected by monitoring the system (or simulated system) performance or by reading the code. Furthermore, it is assumed that each error is sufficiently well investigated so that it can be classified as hardware, software, operator, or unknown, and that the unknown category is small, say less than 20%.

#### 3.0 DEFINITION OF SOFTWARE RELIABILITY

#### 3. 1 Definition

A definition for software reliability is given below in keeping with the factors discussed above. This definition is a slight modification of one given by Hesse: Software reliability is defined as the probability that a given software program operates for some time period, without software error, on the machina for which it was designed given that it is used within design limits. Once we have related reliability to a probability, as in the above definition, the mathematical basis of the measure is well founded. Of course, the problems in interpreting terms such as error and design limits still exists.

#### 4.0 ERROR DATA AND MODELS

#### 4. 1 Error Data

As is the case with all studies, data is difficult and generally costly to obtain. Good and complete records are not kept in most situations. Record keeping is generally better in the case of large military and space programs and after a release of a large commercial operating system. Although a fairly large amount of such data exists, military secrecy and industrial proprietary policies inhibit its publication in many cases. Some of this data which has been published appears in Refs. 2, 5, 14. (Also see paper by Shooman and Bolsky in Ref. 6).

Assume that a typical operating system for a large computer is undergoing continual develop-ment and that new features and capabilities are being added. The manufacturer's development group deals with a continually changing product, but external versions (generally called releases) are only made available periodically, say every 6 months. Although the manufacturer tries to thoroughly test each release, the exercising of the program by a fair proportion of the large and diverse user community is more comprehensive than any test he can devise. Consequently, soon after release of a new version, the number of errors found per month (error rate) rises rapidly to a peak. As these are diagnosed and corrected, the number of residual errors decreases and the error rate begins to decrease. When a new release is distributed, this behavior is repeated. Detailed data on the normalized number of errors since release for three different supervisory systems (operating systems) is given in Fig. 4-1. Note that the horizontal axis units are months of debugging 7. In this case 7 is identical with operating time, t; however, this is not always the case.

Note that the shapes depicted in Fig. 4-la, b, c vary. If we assume that the number of remaining errors decreases monotonically and that the error discovery rate is proportional to number of remaining errors, exponential decay is obtained. This explains in a gross way the "tail" of the curves. The initial behavior may be due to the fact that initially only a few installations are using the new release, and it is not until a few months later that a sizeable proportion of users have instituted this software. Thus, it might be more appropriate to let  $\tau$  represent a more general resource variable such as user-months, or to serve as a more realistic horizontal scale parameter. (See Ref. 3 for a more detailed discussion of these curve shapes.)

#### 4.2 Error Model

Referring to the data discussed in the previous section we see that although the curve shapes differ, the vertical and horizontal scales are similar. Based on this result we can proceed to formulate a general error model using the number of machine language instructions I<sub>T</sub> as a normalizing factor. (Sec. 4. 3 discusses the possibility that the total number of operators and operands is a better normalizing factor.)

Basically, the error model used in this paper assumes that the total number of errors in the program is fixed and that if we record the cumulative number of errors corrected during debugging, then the difference represents the remaining errors. The following section on reliability models will relate the probability of encountering a software bug to the number of residual bugs.

The normalized error rate is defined as

 $\rho(\tau)$  = errors/total number of instructions/ month of debugging time. (4-1)

Thus, Figs. 4-1 a, b, c are plots of p(r) vs. r.

Since we are interested in the total number of errors removed, we will define a cumulative error curve,  $\varepsilon(\tau)$ , which is the area under the  $\rho(\tau)$  curve:

 $\varepsilon(\tau) = \int_{0}^{\infty} \rho(x) dx = \text{cumulative errors/total}$ so mumber of instructions
(4-2)

and  $\rho(\tau)$  is of course the slope of the  $\varepsilon(\tau)$  curve:

 $\rho(\tau) = de(\tau)/d\tau \tag{4-3}$ 

A curve of the cumulative error data for the supervisory system A of Fig. 4-1 is shown in Fig. 4-2. If similar curves for  $\varepsilon(\tau)$  were drawn for the other examples of Figs. 4-1 all would start at zero, increase slowly, then move rapidly, and finally, more slowly approaching a slowly increasing or zero rate. Because of this similarity in behavior, a cumulative curve such as  $\varepsilon(\tau)$  is not too useful to depict differences in behaviors; thus, the derivative curve,  $\rho(\tau)$ , is more useful for this purpose. Both curves are needed for a detailed study.

If we assume that the total number of errors in the program E<sub>T</sub> is constant and that the program contains I<sub>T</sub> total instructions and that no new errors are added during debugging, then the

asymptote which the  $\varepsilon(\tau)$  curves approach is  $E_{T}I_{T}$ . If we assume that all detected errors are corrected errors, then by inspection of Fig. 4-2, we can write an expression for the number of residual errors:

$$\epsilon_{\mathbf{r}}(\tau) = (\mathbf{E}_{\mathbf{T}}/\mathbf{I}_{\mathbf{T}}) - \epsilon_{\mathbf{c}}(\tau)$$
 (4-4)

We assume that in any sizeable program it is impossible to remove all errors, so:  $\varepsilon_{\rm c}(\tau) < E_{\rm T}/I_{\rm T}$  and  $\varepsilon_{\rm r}(\tau) > 0$ . Also since we assume that most programs eventually reach a reasonable debugged state, we may assume that for large  $\tau$ ,  $\varepsilon_{\rm r}(\tau)I_{\rm T}/E_{\rm T}$  is small.

In order to test the hypothesis that the normalized behaviors of  $\varepsilon_{\mathbf{r}}(\tau)$  and  $\rho(\tau)$  hold for a wide variety of program sizes we make the following comparisons with the data in Fig. 4-1: (1) In order to test the hypothesis that the normalized number of errors ET/IT is somewhat constant for a variety of programs, we compute the ratio and compare the results. (2) An allied hypothesis is that debugging proceeds at a roughly similar average rate po over an entire project. The results are given in Table 4-1. The value of ET/IT varies about the average by +48% and -31% and that of po varies about the average by +75% and -31%. Note that all these programs are about 1/4 million machine language statements in size. We now present similar data for small programs in Table 4-2. In this case both the values of  $E_T/I_T$  and  $p_0$  vary about the average by +79% and -36%.

The data in Table 4-2 includes data taken during module test as well as during integration testing and as might be expected (because of the two phases being lumped), the average value of  $E_{\rm T}/{\rm IT}$  for the small programs data is 2.15 times larger than the large program data whereas the value of  $\rho_0$  is 1.53 times larger. Drawing these various facts together allows us to state that within a factor of perhaps 2, the values of  $E_{\rm T}/{\rm IT}$  and  $\rho_0$  seem to be constant for a wide variety of programs.

One further comment is in order before we leave the subject of error models. Some experienced programmers have challenged the assumption that no new errors are generated during debugging. A newly devised model, formulated by this author and his co-werkers describes error generation and is discussed and developed in Reference 7.

#### 4.3 Program Length and Program Complexity

There are many similarities between natural and computer languages, and we will make use of the analogies between nouns and verbs and operands and operators to obtain better length measures of a program based on Zipf's law applied to programs. 14, 15

Zipf studied the relationship between relative frequency of occurrences  $f_{r}$ , (actually the number of occurrences  $n_{r}$  of rank r words in a sample of size n), and rank r for words from English, Chinese and the Latin of Platus<sup>8</sup>.

Careful study of Zipf's data and that of others shows that  $f_T$  vs. r plots as a straight line on loglog paper, with a unity slope, thus we arrive at

the simple relationship called Zipf's law,

$$f_{\mathbf{r}} \cdot \mathbf{r} = \frac{n_{\mathbf{r}}}{n} \cdot \mathbf{r} = c \tag{4-5}$$

The constant c can be interpreted as the relative frequency of the rank 1 word type. (also the intercept with the r = 1 line).

Solving for n<sub>r</sub> and summing both sides of Eq. 4-5 for the t different word types we obtain \*

$$n = \sum_{r=1}^{t} n_r = cn \sum_{r=1}^{t} \frac{1}{r}$$
 (4-6)

The summation of the series 1/r is given by

$$\sum_{r=1}^{t} \frac{1}{r} = 0.5772 + \ln t + \frac{1}{2t} - \frac{1}{12t(t+1)} - \dots$$
 (4-7)

Substitution from Eq. (4-7) into Eq. (4-6) (retaining only 2 terms for modest size t) yields

$$n \approx cn (0.5772 + ln t)$$
 (4-8)

We can eliminate the constant c from Eq. (4-8) by considering the behavior of Zipf's law for the smallest rank which is where r<sub>max</sub>=t(eg. if there are 100 types, then the largest rank is obviously 100). In most cases the rarest type (largest rank) will occur only once, thus,n<sub>\_\_</sub>=1. Substituting these values yields, c=t/n, which when combined with Eq. (4-8) gives

$$n = t(0.5772 + ln t)$$
 (4-9)

The results to date have shown that both operators, operands, and the sum of operators plus operands fit Zipf's law fairly well? Thus, instead of using  $I_T$  as a normalizing factor in Eq. (4-1) to (4-4) one could use total word length n.

# 4.4 Estimation of Program Length Early in Design

A method of initially estimating program length 9 is to assume the analyst initially has a complete description of the problem and that a partial analysis and choice of key algorithms has been made. An elementary approach might be to estimate the token size by (1) Estimating the number of operator types which will be used in the language by the assigned programmers. (2) Estimating the number of input variables, output variables, intermediate variables, and constants needed. (3) Sum the estimates of step (1) and (2) and substitute for t in Eq. (4-9).

# 5.0 RELIABILITY MODELS

#### 5. l Basic Assumptions

We assume that operational software errors occur due to the occasional traversing of a portion of the program in which a hidden software bug is lurking. We begin by writing an expression for the probability that a bug is encountered in the time interval  $\Delta t$  after t successful hours of operation. This must be proportional to the probability that any randomly chosen instruction contains a bug, i. e., the fractional number of remaining bugs  $\varepsilon(\tau)$ .

\*Most frequent word type is rank le second most frequent is rank 2, least frequent is rank t.

From a study of basic probability and reliability theory,  $^{10}$  we learn that the probability of failure in time interval t to t + \Dt given that no failures have occurred up till time At is proportional to the failure rate (hazard function z(t), and the reliability is related to the failure rate

$$P(t < t_f \le \underline{t} + \Delta t | t_f > \underline{t}) = z(t)\Delta t = K \varepsilon_r(\tau) \Delta t$$
(5-1)

where t<sub>f</sub> = operating time to failure, (occurrence of a software error)

 $P(t < t_f \le t + \Delta t | t_f > t) = probability of failure$ in interval  $\Delta t$ , given K=anarbitrary constant no previous failure.

$$R(t) = e^{-\int_0^t z(x) dx}$$
(5-2)

Note that in Eq. 5-1 two time variables appear: first there is t the operating time in hours of the system and second there is r the debugging time in months (or more generally, the debugging resource variable). Once the assumptions in Eq. 5-1 have been made, the reliability and mean time to failure functions follow directly.

#### 5.2 Reliability Model

By combining Eqs. (5-1), (5-2), and (4-4) and assuming that K and  $\varepsilon_r(\tau)$  are independent of operating time t, we obtain for the reliability function

ing time t, we obtain for the reliability function 
$$-\left[K\varepsilon_{\mathbf{r}}(\tau)\right]t - K\left[\frac{E_{\mathbf{T}}}{I_{\mathbf{T}}} - \varepsilon_{\mathbf{c}}(\tau)\right]t$$

$$R(t) = e^{-\gamma t} = e^{-\gamma t}$$
Basically the above equation states that the

Basically the above equation states that the probability of successful operation without software bugs is an exponential function of operating time. When the system is first turned on, t=0 and R=1. As operating time increases the reliability monotonically decreases as shown in Fig. 5-1. We depict the reliability function for three values of debugging time,  $\tau_0 < \tau_1 < \tau_2$ . From this curve we may make various predictions about the system reliability. For example, looking along the vertical line t=1/y we may state:

- 1. If we spend  $\tau_0$  hours of debugging, then  $R(1/\gamma) = 0.35$
- 2. If we spend  $\tau_1$  hours of debugging, then  $R(1/\gamma) = 0.50$
- 3. If we spend 72 hours of debugging, then  $R(1/\gamma) = 0.75$

#### 5.3 MTTF Model

A simpler way to summarize the results of the reliability model is to compute the mean time to (software) failure, MTTF by using the formula

MTTF = 
$$\int_0^{\infty} R(t)dt = \frac{1}{K\varepsilon_{\mathbf{r}}(\tau)} = \frac{1}{K\left[\frac{E_{\mathbf{T}}}{I_{\mathbf{T}}} - \varepsilon_{\mathbf{c}}(\tau)\right]}$$
 (5-4)

If we let  $\rho(\tau)$  be modeled by a constant rate of error correction  $\rho_0$  (see Ref. 3 for other models),

MTTF = 
$$\frac{1}{K\left[\frac{E_T}{I_T} - o_o\tau\right]} = \frac{1}{8(1-\alpha\tau)}$$
 (5-5)

where 
$$g = \frac{E_T}{I_T}$$
 K and  $\alpha = \frac{\rho_o I_T}{E_T}$ 

In Fig. 5-2, 3 x MTTF is plotted vs. ar. We see that the most improvement in MTTF occurs during the last 1/4 of the debugging.

Other similar models have been proposed and the reader is referred to the literature: Jelinski and Moranda 11, Schick and Wolverton 12.

#### 5. 4 Experimental Reliability Data

If we had just deployed a large hardware-software system for field use, we could monitor its reliability by carefully recording the operating time and documenting each failure in detail. Thus, we could obtain the times between failure. Investigation of each failure should allow one to classify all failures as hardware, software, operator, or unknown. If we segregate the software times between failure and plot their average week by week, we will have a quantitative measure of operational software reliability. We would expect the operational MTTF to increase for the first month (year, in some cases) or so as software bugs detected in service are removed, then gradually to level off to a relatively constant value. This is, of course, an after-the-fact evaluation of the software design and does not allow one to measure progress and/or need for improvement of the software design while it is under development.

The earliest stage at which an entire system can be functionally tested is during system integration using the system exerciser (functional test) program. If this test is performed at the beginning of system integration, the result will be a succession of very short runs and immediate crashes. Most software test personnel would instinctively comment that this is as expected since the system is still in "poor shape" and such a test should be delayed until the end when the system is in "good shape". A bit of reflection leads one to the conclusion that it is just this frequent crashing which leads to a quantitative assessment of the poor initial reliability.

We now focus on the test data and how it should be analyzed. The necessary information which must be recorded for each run of the system test program is how long the test ran, whether an er-ror occurred, and if the error is a software error. Sufficient dumps and other documentation must be recorded for subsequent analysis in order to segregate errors into hardware, software, operator, etc., errors. Each of the r successful runs represent T1, T2, ... Tr hours of success. If there are n total runs, then each (n-r)=x unsuccessful run represents t<sub>1</sub>, t<sub>2</sub>,...t<sub>n-r</sub> successful run hours before failure. The total number of successful run hours H is given by

$$H = \sum_{i=1}^{r} T_{i} + \sum_{i=1}^{n-r} t_{1}$$
 (5-6)

Assuming that the failure rate is constant, we denote it by \ and compute it as the number of failures per hour, along with its reciprocal which is the MTTF (for a constant failure rate, c.f. Eqs. (5-3) and (5-4))

Failure Rate = 
$$\lambda = \frac{x}{H}$$
 (5-7)

of

$$MTTF = \frac{1}{\lambda} = \frac{H}{x}$$
 (5-8)

Since we are interested in software failures, we assume that the outputs as well as dumps are carefully investigated for the x failures, and the failures are divided into  $\mathbf{x}_h$  hardware failures.  $\mathbf{x}_s$  software failures,  $\mathbf{x}_o$  operator failures, and x unknown failures. (Hopefully  $\mathbf{x}_u/\mathbf{x}$  will be small. Then the software failure rate and MTTF are defined by Eqs. (5-7) and (5-8) with  $\mathbf{x}_s$  substituted for  $\mathbf{x}$ .

Based upon the results of Sec. 5. 3 and 5. 4 we can calculate the model constants. We assume a known program size and careful collection of error data, then I<sub>T</sub> and  $\varepsilon_{\rm C}(\tau)$  are known values and only the constants K and E<sub>T</sub> remain to be determined. These two unknowns K and E<sub>T</sub> can be evaluated by running a functional test after two different debugging times,  $\tau_1 < \tau_2$  chosen so that  $\varepsilon_{\rm C}(\tau_1) < \varepsilon_{\rm C}(\tau_2)$ . We then equate Eqs. (5-4) and (5-8) at times  $\tau_1$  and  $\tau_2$ 

$$\frac{H_1}{\kappa_{s_1}} = \frac{1}{\kappa \left[ \frac{E_T}{T_T} - \varepsilon_c(\tau_1) \right]}$$
 (5-9)

$$\frac{H_2}{x_{s_2}} = \frac{1}{K \left[ \frac{E_T}{I_T} - \epsilon_c(\tau_2) \right]}$$
 (5-10)

Taking the ratio of Eq. (5-9) to (5-10) and using Eq. (5-8) yields

$$\hat{E}_{T} = \frac{I_{T} \left[ \left( \lambda_{s_{2}} / \lambda_{s_{1}} \right) \epsilon_{c}(\tau_{1}) - \epsilon_{c}(\tau_{2}) \right]}{\left( \lambda_{s_{2}} / \lambda_{s_{1}} \right) - I}$$
(5-11)

Once  $\hat{E}_T$  has been computed from Eq. (5-11), we obtain  $\hat{K}$  by substituting Eq. (5-11) into (5-9) which yields

$$\hat{K} = \lambda_{s_1} / [(\hat{E}_T / I_T) - \epsilon_c(\tau_1)] \qquad (5-12)$$

The "hats" above ET and K in Eqs. (5-11) and (5-12) denote estimates of the parameter. Further discussions of this parameter estimation technique, the more powerful maximum likelihood estimation technique, and accuracy questions appear in Ref.13.

#### 5.6 Verification of the Model

Two papers have appeared in the literature which apply the above model or extensions of it to practical data. The paper by Miyamoto in Ref. 6 reports results very similar to Fig. 5. 2. In a more extensive study of four program developments Musa 19 reported good agreement between MTTF estimated during development (c. f. Eqs. (5-5), (5-11), (5-12) and subsequent operational measurements.

# 6.0 MICRO RELIABILITY MODELS

6.1 Introduction

The software reliability prediction model of Sec. 5 concentrated on the bulk (macro) aspects of a program. This section describes a newly developed micro model 16 which is based on program structure.

It is assumed that the program has been written in structured or modular form so that decomposition into its constituent parts is simple. Further, we assume that via analysis of the program the decomposition can be related to several paths or other functional structures within the program.

The model is constructed based upon the frequencies with which each of the j paths are run,  $(f_i)$ , the running time of each path,  $(t_j)$ , and the probability of error along each path,  $(q_i)$ .

#### 6.2 Development of the Model

The model is developed by calculating an expression for the number of failures,  $n_{\rm f}$ , experienced in N tests of the system, and the number of test hours H. (see Ref. 14). Then the system failure rate  $z_0$  is computed from

$$z_0 = \lim_{N \to \infty} \frac{n_f}{H}$$
 (6-1)

The term N appears in numerator and denominator and Eq. (6-1) yields in the limit

$$z_{o} = \frac{\sum_{j=1}^{i} f_{j}q_{j}}{\sum_{j=1}^{i} f_{j}(1 - \frac{q_{j}}{2})t_{j}}$$
(6-2)

Equation (6-2) can be substituted into Eq. (5-2) to obtain a reliability model. Measurement of the model parameters is discussed in Ref. 16.

#### REFERENCES

- M. Shooman, "Software Reliability: Analysis and Prediction." published in "Generic Techniques in Systems Reliability Assessment", Ed. by E. J. Henley and J. W. Lynn, Noordhoff Intl. Publ., Reading, Mass., 1976.
- S. J. Amster and M. L. Shooman, "Software Reliability: An Overview, "Reliability and Fault Tree Analysis, p. 655, ed. R. Barlow et al., Society for Industrial and Applied Math., Philadelphia, Pa., 1975.
- M. Shooman, "Probabilistic Models for Software Reliability Prediction, "Conference on Statistical Methods for the Evaluation of Computer System Performance, Brown Univ., Nov. 1972. Published in "Probabilistic Models for Software, " Freiberger Editor, Academic Press, New York, N. Y., 1972, pp. 485-502.
- Proceedings, 2nd Intl. Conference on Software Engineering, Oct. 13-15, 1976, IEEE Computer Society, New York.

- J. Hesse, A. Kientz, J. Dickson and M. Shooman, "Quantitative Analysis of Software Reliability, 1972 Annual Reliability Symposium Proceedings, IEEE, New York, N. Y., January 1972.
- Proceedings of the Intl. Conference on Reliable Software, "Los Angeles, April 21-23, 1975, IEEE Cat. No. 75 CH0940-7CSR.
- M. L. Shooman and S. Natarajan, "Effect of Manpower Deployment and Error Generation on Software Reliability, "Proceedings of the MRI Symposium on Computer Software Reliability, April 1976. Available from Polytechnic Press, Polytechnic Institute of New York.
- George K. Zipf, "The Psycho-Biology of Language: An Introduction to Dynamic Philology,"
   M. L. T. Press, 1965. (First Houghton Miffin Edition, 1935).
- A. Laemmel and M. Shooman, "Statistical (Natural) Language Theory and Computer Program Complexity," Internal. Report, Polytechnic Institute of New York, Winter 1976.
- M. Shooman, "Probabilistic Reliability: An Engineering Approach," McGraw-Hill Book Co., New York, N. Y., 1968.
- J. Jelinski and P. B. Moranda, "Software Reliability Research," same source as Ref. 3.
- G. J. Schick and R. W. Wolverton, "Assessment of Software Reliability," 11th Annual Meeting, German Operations Research Society, Hamburg, Germany, Sept. 1972.
- 13. M. Shooman, "Operational Testing and Software Reliability Estimation During Program Developments," Record of "1973 IEEE Symposium Computer Software Reliability," IEEE, New York 1973.
- 14. M. L. Shooman, "Software Engineering: Reliability, Design, Management," Notes for Course EE909, Polytechnic Institute of N. Y., 1976.
- M. Halstead, "Software Physics: Basic Principles," IBM Research Report RJ1582, IBM Research, Yorktown Heights, N. Y., May 1975.
- 16. M. L. Shooman, "Structural Models for Software Reliability Prediction," Proceeding of the 2nd International Conference on Software Engineering, Oct. 1976, San Francisco, Calif., IEEE Computer Society, New York.
- 17. Proceedings of the 1st Intl. Conference on Software Engineering, IEEE Computer Society, Sept. 11-12, 1975, New York.
- F. P. Brooks, "The Mythical Man-Month: Essays on Software Engineering," Addison-Wesley, Reading, Mass., 1975.
- 19. IEEE Trans. on Software Engineering, IEEE Computer Society, New York City (First published in 1975.) Paper by John Musa, "A Theory of Software Reliability and Its Applications, "Vol. SE-1, No. 3, Sept. 1972.
- Fumio Akiyama, "An Example of Software System Debugging," IFIP Congress 71, Ljubljana, Yugoslavia, Aug. 1971.
- Barry W. Boehm, "Software and Its Impact: A Quantitative Assessment, "Datamation, May1973.

Table I-I

	Failure of	Pailures
	( CPU	7
lardvare	Memory	12
	Disk pack	1
	Fixed head disk	2
	Power	3
	Miscellaneous	4
Software	Operating system	19
	Applications programming	8
	Totals	56

	Failure of	failure rate
	( CPT	4.08 × 10"
Hardware	Memory	7.00
	Disk pack	.59
	Pixed head disk	1.18
	Power	1.77
	Miscellaneous	2.32
Software	Operating system	11.1
	Applications programming	4.6
	Totals	32.8 × 10"

	Pailure of	S of Total
	( CPU	12.5
	Memory	21.4
	Disk pack	1.8
Hardware	Fixed head disk	3.6
	Power	5.4
	Miscellaneous	7.1
Software	Operating system	34.0
	Applications programming	14.0
	Totals	100 \$

Bardware and software failure rates (see Ref. 1)

TABLE 4-1
Computation of Model Constants from the Data of Ref. 5

Program	Size	E <sub>T</sub> /I <sub>T</sub>	Po
Supervisory A	210 K	$6.14 \times 10^{-3}$	0.875 x 10-3
Supervisory B	240	7.97	0.996
Supervisory C	230	7.48	1.25
Application A	240	13.20	2.20
Application B	240	7.70	1.54
Application C	240	7.00	1.00
Application D	240	12.90	0.995
Average		. 8. 92	1 26

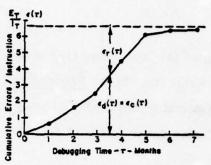


Fig. 4-2. Cumulative error curve for supervisory system A given in Fig. 4-1.

TABLE 4-2
Computation of Model Constants from the Data of Ref. 20

Program	Size	E <sub>T</sub> /I <sub>T</sub>	Po
MA	4. 03 K	25.4 x 10 <sup>-3</sup>	2.54 x 10 <sup>-3</sup>
MB	1.32	13.7	1.37
MC	5. 45	17.1	1.71
MD	1.67	15.6	1.56
ME	2.05	34.6	3.46
MF	2.51	14.7	1.47
MT	2.10	12.4	1.24
MG	0.70	22.9	2.29
MH	3.79	13.2	1.32
MX	3.41	23.4	2.34
Average		. 19.3	1.93

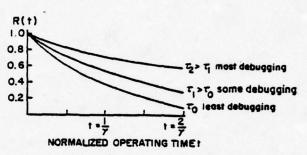
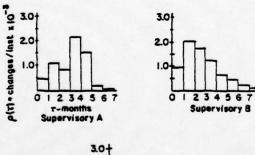


Fig. 5-1. Variation of reliability function R(t) with debugging time  $\tau$ .



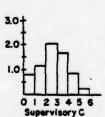


Fig. 4-1. Normalized error rate versus debugging time for three supervisory programs.

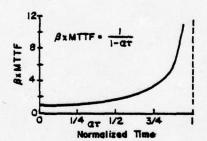


Fig. 5-2. Comparison of MTTF vs. debug time for the constant error debug rate model.

# ACKNOWLEDGMENT

This paper originally appeared under the title, "The Spectre of Software Reliability and Its Exorcism" in the <u>Proceedings of the 1977 Joint Automatic Control Conference</u>. This material is in part based upon research work supported by the Rome Air Development Center under Contract No. F-30602-74-C-0294.

## SOFTWARE RELIABILITY MEASUREMENT\*

John D. Musa

Bell Laboratories Whippany, New Jersey 07981

#### ABSTRACT

The quantification of software reliability is needed for the system engineering of products involving computer programs and the scheduling and monitoring of software development. It is also valuable for the comparative evaluation of the effectiveness of various design, coding, testing, and documentation techniques. This paper outlines a theory of software reliability based on execution or CPU time, and a concomitant model of the testing and debugging process that permits execution time to be related to calendar time. The estimation of parameters of the model is discussed. Application of the theory in scheduling and monitoring software projects is described, and data taken from several actual projects are presented.

#### I. INTRODUCTION

Three important current needs in the field of software engineering have been instrumental in stimulating work (and more particularly, the author's work) in developing a way to measure software reliability:

- (a) an urgent requirement to quantify reliability due to its importance as a parameter in the engineering of software systems
- (b) the need for better methods of establishing and monitoring schedules for software development projects, and
- (c) the necessity of evaluating the effectiveness of different software engineering techniques.

The growing trend in computing toward interactive data processing systems, both time-sharing and transaction-based, has resulted in an increasing focus on the reliability of such systems, because of the greater operational and cost impacts of real time malfunctions. For example, consider the effects of breakdowns of airline reservations, banking, or stock broker-

age online systems. The concurrent trends toward distributed processing and networking have increased the scope, complexity, and interdependence of computing systems, multiplying the risk of failure and therefore further reinforcing the concern over reliability.

Measurement becomes important as soon as one recognizes that in software as in hardware there can be too much as well as too little reliability. Improvement of reliability, of course, costs money, and usually impacts development schedules and system performance (in the case of software, through increased memory, processing time, and peripherals requirements). The system engineer and the manager have to make design tradeoffs between reliability. performance. schedules and other factors, either implicitly or explicitly. There is a pressing need for good tools that will permit the decisions to be made explicitly. Such tools exist in the case of hardware, but there has not been a corresponding development in the area of software. However, software is an and usually the essential element in the systems we have been describing. Consequently, there is an urgent need to find ways to characterize and measure software reliability and to develop methods of combining reliabilities of software and hardware elements to establish system reliability.

Presented at the Software Life Cycle Management Workshop, Airlie, Virginia, August 22-23, 1977.

<sup>• 1977,</sup> Bell Telephone Laboratories, Inc.

A second issue of concern in the data processing community is how to schedule a software development project and monitor its progress. This task becomes both most critical [1] and most difficult during the latter phases of the project where testing and removal of errors is the primary activity. The time required for testing is mainly dependent on the initial reliability of the design and code and the reliability goal to be attained. Fred Brooks [2] points out the uselessness of "man month estimating," particularly in the component test and system test phases, because of the sequential nature of debugging. He also emphasizes the criticality of the system test phase to a project, and the need for better estimating techniques.

Testing progress has been generally very difficult to evaluate. Managers have used methods such as intuition of designers or test team, percentage of tests completed, and the successful execution of critical functional tests. None of these have been completely satisfactory and some have been most unsatisfactory. On the other hand, reliability [or better, present mean time to failure (MTTF)] offers substantial promise of being a good monitoring metric.\*

Third, the computing field has seen and continues to see a plethora of new software engineering techniques: requirements definition approaches, design techniques, coding methods, testing techniques, and documentation methodologies. Unfortunately, there has been little quantitative evaluation of these techniques. The initial enthusiasm that greeted such innovations (due perhaps to the general recognition of software engineers of the need to develop their technology) soured and turned to skepticism as many of the ideas turned out to be not sufficiently effective to justify their cost. The inability of managers and practitioners to differentiate between good and bad new techniques has led to a general resistance to change

that is counterproductive. Finding a way to measure reliability offers promise of establishing at least one criterion for the evaluation of new technology. For example, you would then be able (in an experiment) to concretely determine the increase in MTTF at the start of system test that results from using structured programming. You could compare the dollar value of this benefit with the increased cost required in the design phase.

The objective of the main body of this paper is to give an overview of a software reliability theory developed by the author and its application. Additional details that will be useful to those who wish to apply the theory are contained in the Appendices. A brief glossary of terms is provided in Appendix A.

## II. SOFTWARE RELIABILITY VS. HARDWARE RELIABILITY

Software reliability will be defined like hardware reliability; i.e., it is the probability of failure-free operation in a specified environment for a specified time. A "failure" is defined as an unacceptable departure of program operation from program requirements. An "error" is the software defect that causes the failure. The concept of software reliability differs from that of conventional hardware reliability in that failure is not due to a wearing-out process. Once an error is properly fixed, it is, in general, fixed for all time. Failure usually occurs only when a program is exposed to an environment that it was not designed or tested for. For most programs, in practice, the large number of possible states and inputs makes perfect comprehension of the program requirements, perfect implementation and complete testing generally impossible. Thus, software reliability is essentially a measure of the confidence we have in the design and its ability to function properly in its expected environment. Although the failure mechanism is different for software than hardware, the result for the system is the same; this is why a theory that is compatible with hardware reliability can be developed.

In some cases there may be a more natural unit. For example, in transaction processing systems where one can define a "standard" common transaction, such as a reservation for airline reservation systems, one may wish to talk in terms of reservations processed between failures rather than MTTF. The conversion is easily made.

#### III. BASIC CONCEPTS

Jelinski and Moranda [3] developed one of the earliest software reliability models. They postulate an error detection rate that is proportional to the number of errors remaining. Miyamoto [4] has published data confirming this relationship. Shooman [5], [6] used a similar model, relating error detection rate also to the program size and instruction processing rate. He investigated various models for error correction and proposed a two-point parameter estimation approach. Schneidewind [7],[8] took an empirical approach and suggested a reliability prediction scheme based on fitting failure intervals with an appropriate reliability function. He has developed models for error detection and correction processes and applied maximum likelihood estimation to the determination of the parameters of these processes [10]. Littlewood and Verrall [11], [12] developed a very general Bayesian reliability growth model for software in which repair actions diminish the failure rate in a probabilistic rather than deterministic fashion.

After considerable study, based partly on experience with software development projects, the author concluded that a superior reliability model, incorporating some of the concepts of the foregoing models, could be constructed if one first thought in terms of execution time, i.e., the actual processor time utilized in executing the program, and then determined how execution time and calendar time were related. Execution time appeared to be the best practical measure for characterizing the stress placed on software. Furthermore, one could relate the error correction rate to the instantaneous failure rate during test, eliminating the need for developing an error correction model. Finally. execution time could be related to calendar time in a simple and straightforward fashion because the relationship between the two is generally paced at any given time by one of the limiting resources: following failure identification personnel, failure correction personnel, or computer time. The failure identification personnel are usually members of a test team and the failure correction personnel are usually the original designers.

Included in the theory is a statistical procedure for continually re-estimating the parameters of the reliability model during a test phase of a software project. Using the time intervals between failures experienced during this phase, one can estimate:

- the total number of failures possible during the maintained life of the software (and the associated number of errors),
- (b) the initial MTTF before debugging started.
- (c) the present MTTF,
- (d) the additional number of failures (or errors) that must be detected in order to reach whatever MTTF objective has been established for the project, along with the additional number of computer hours of testing required, and
- (e) the additional calendar time in days required to meet the MTTF objective.

All of these estimated quantities are generated in terms of a maximum likelihood value and confidence intervals (typically 50, 75, 90, and 95 percent).

# IV. EXECUTION TIME COMPONENT OF MODEL

The execution time model is based on certain assumptions that appear to be met reasonably well by most executable programs and most development projects, assuming a properly planned and executed test effort:

Assumption 1: Tests are representative of the environment in which the program will be used and are continuously global.

Assumption 2: Failures are independent of each other and distributed at any time with a constant average execution time occurrence rate that is proportional to the number of errors remaining.

Assumption 3: All failures are observed.

Assumption 4: The error correction rate is proportional to the failure detection rate at all times.

The discussion of these assumptions that follows is supplemented by details in Appendix B.

It is not necessary that testing exhaust all possible environmental or input conditions; however, test input sets should be selected randomly from the complete set of input sets. By

"continuously global" it is meant that the group of tests that occupy a small testing interval (perhaps a calendar day or so) exercise essentially all the code of the program. If tests are selected so that only module A is tested on day 1 and module B is tested on day 2, estimates made on either day will not be representative of the program. If the tests cannot be "continuously global" because testing starts before integration is complete or because one component (e.g., the operating system) is thoroughly tested before the others are, the situation is analogous to the addition of design changes. See Appendix C for methods of handling this condition.

Note that the constant execution time occurrence rate postulated in Assumption 2 implies that failures constitute a piecewise Poisson process (the parameter changes whenever an error is corrected) with execution time between failures distributed piecewise exponentially. Studies in progress using real project data appear to be supporting this assumption, to a first approximation.

Assumption 3 is necessary because a failure can sometimes be subtle. Subtle failures can occur for hardware also, but probably less frequently than for software. If some failures are missed, estimation of the program's reliability will be optimistic. The author found that one is more likely to miss failures in the operational period of the program than the test period since heightened attention to failures is characteristic of most test periods. Hence, the estimate of the program's reliability is still relatively pessimistic, which is satisfactory from a practical viewpoint.

The determination of which departures of operation from program requirements are "unacceptable" and therefore failures is up to the user or system engineer, as long as the determination is made consistently throughout a project. For example, one may only consider program crashes requiring interruption of processing and program reload as unacceptable. On the other hand, any situation in which system requirements are not met, such as an incorrect title on a report, may be considered as a failure. One may even wish to classify failures by severity and obtain reliability measures for each class.

In Assumption 4, the constant of proportionality between the error correction rate and the failure detection rate is called the error reduction factor B. It can account for:

- (a) error growth due to new errors that are spawned in the process of correcting errors or that emanate from the added or modified code resulting from design changes,
- (b) errors found by reading that was stimulated by the detection of a related error during test, and
- (c) a proportion of failures whose causative errors cannot be determined and hence cannot be corrected.

The latter situation may exist because insufficient data have been gathered to track the causes of the failures and the failures cannot be reproduced. This state of affairs often occurs for operating systems in computation centers. See Appendix D for more details.

Based on the assumptions that have been made, it has been shown [12] that the net number of errors detected and corrected n is an exponential function of the execution time  $\tau$ :

$$n = N_0 \left[ 1 - \exp \left[ -\frac{C\tau}{M_0 T_0} \right] \right]. \tag{1}$$

where  $N_0$  is the number of inherent (unreduced by debugging) errors,  $M_0$  is the total number of failures possible during the maintained life of the software,  $T_0$  is the MTTF at the start of test, and C is the testing compression factor. "Maintained life" refers to the period extending from the start of test to the discontinuance of program failure correction. Note that  $M_0$  is also the number of failures that must be experienced to remove all errors. The testing compression factor indicates the extent to which operational runs occurring under duplicate conditions have been reduced or eliminated during test by test selection and design. For example, one hour of test may represent 12 hours of actual operation. Thus C represents the ratio of equivalent operation time to test time. A more detailed discussion is given in [12].

Now it will be seen that failures and errors are related through

$$N_0 = BM_0 \tag{2}$$

and

$$n = Bm , (3)$$

where m is the number of failures experienced. Expressions in terms of failures are most useful, since they are most readily measurable. Hence from (1), (2), and (3) we obtain

$$m = M_0 \left[ 1 - \exp \left[ -\frac{C\tau}{M_0 T_0} \right] \right]$$
 (4)

The present MTTF T is then shown to be

$$T = T_0 \exp\left[\frac{C\tau}{M_0 T_0}\right]. \tag{5}$$

MTTFs are measured in execution time. Note that the present MTTF increases as testing proceeds. Reliability R for an operational period  $\tau'$  is given by

$$R = \exp\left[-\frac{\tau'}{T}\right], \qquad (6)$$

By eliminating  $\tau$  between (4) and (5), solving for m, evaluating the resultant equation at  $m_1$ ,  $T_1$  and  $m_2$ ,  $T_2$ , and subtracting, we obtain an equation for the number of failures  $\Delta m$  that must be experienced to increase MTTF from  $T_1$  to  $T_2$ :

$$\Delta m = M_0 T_0 \left[ \frac{1}{T_1} - \frac{1}{T_2} \right].$$
 (7)

If (5) is solved for  $\tau$  and evaluated at  $\tau_1$ ,  $T_1$  and  $\tau_2$ ,  $T_2$  and the resultant equations are subtracted, we obtain the relationship that specifies the additional execution time  $\Delta \tau$  required to increase the MTTF from  $T_1$  to  $T_2$ :

$$\Delta \tau = \frac{M_0 T_0}{C} \ln \left[ \frac{T_2}{T_1} \right]. \tag{8}$$

Data taken on six different development projects indicates that the model, characterized by (4), is followed quite well. An example is shown in Fig. 1, where number of failures is plotted against execution time. The vertical axis has been transformed so that data that fit (4) plot as a straight line (statistical variation around the line is expected and must be allowed for). It will be observed that the fit is quite good. Similar results have been obtained on other projects. It should be noted that changes in  $M_0$  or  $T_0$  due to substantial design changes or substantial amounts of code added will result in a change in the slope of the plot when they occur.

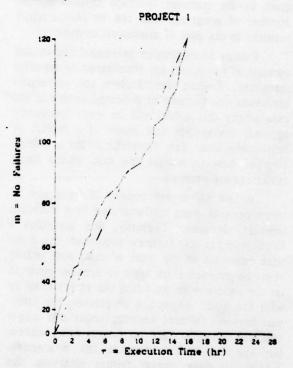


Fig. 1. Verification of execution time model.

# V. CALENDAR TIME COMPONENT OF MODEL

In most projects during test, one is fairly tightly constrained as to the quantities of failure identification (test team) personnel, failure correction (original designer) personnel, and computer time available. Increases in any of these items generally require a long lead time; in fact, as Fred Brooks [2, p. 25] points out, they may actually be counterproductive. Failure identification is the process of comparing test results against program requirements to establish the failures that have occurred. Failure correction is the process of determining the error that caused a failure, removing the error, and proving that the failure no longer occurs. "Computer time" is the measure that is used for allocating computer resources; it is not necessarily the same as execution time. Frequently, it is the residence time of the program in the machine (perhaps divided by the number of programs that can be concurrently resident in the case of multiprogramming).

Failure identification personnel and failure correction personnel are considered as separate resources. Failure identification and correction are usually performed by different people in the case where this model will be most frequently applied, the system test phase of a project of reasonable size. See Appendix D for a discussion of how to handle the case where these resources are merged.

A test effort will consist of from one to three periods, each characterized by a different limiting resource. Typically, one identifies a large number of failures separated by short time intervals at the start of test, and testing must be stopped from time to time in order to let the people who are fixing the errors keep up with the load. As testing progresses, the intervals between failures become longer and longer and the debuggers are no longer fully loaded, but the test team becomes the bottleneck. Finally, at even longer failure intervals, the capacity of the computing facilities is limiting. The debugging process model that is incorporated in this reliability theory utilizes the knowledge of these pacing resources to relate execution time on the computer with the passage of calendar time.

This model makes the following assumptions:

Assumption 5: The quantities of the resources failure identification personnel, failure correction personnel, and computer time that are available are constant for the remainder of the test period.

Assumption 6: Resource expenditures x may be approximated by

$$x = \theta \Delta \tau + \mu \Delta m , \qquad (9)$$

where  $\Delta \tau$  is the increment of execution time,  $\Delta m$  is the increment of failures experienced,  $\theta$  is the partial resource expenditure rate with execution time, and  $\mu$  is the partial resource expenditure rate per failure.

Assumption 7: Failure identification personnel can be fully utilized, computer utilization is constant, and failure correction personnel utilization is established by limitation of error queue length for any debugger. Error queue length is determined by assuming that failure correction is a Poisson process and that servers are randomly assigned in time.

It should be noted that the availability of a fixed quantity of resources does not insure that they are all used. In fact, at any given time, only the limiting resource, at most, will be fully utilized, and the other resources will not.

In justification of Assumption 6, note that identification of failures requires work and computer time that increases with the amount of execution time used and the related amount of test output generated. Also, each failure requires work for verification and determination of its specific nature. Correction of failures requires work and computer time that is proportional to the number of failures. Computer time is expended in both identification and correction of failures; hence, the time used may be expected to depend on both amount of execution time and number of failures.

Assumption 7 implies that the time required to correct a failure is independent of the time required to correct other failures and the probability that an error is fixed during any infinitesimal time period of debugging is equal to the probability that it is fixed during any other such period. Experience has demonstrated that this is a fairly good representation, although it may somewhat overestimate the proportion of errors with short fix times.

The "continuously global testing" assumption made for the execution time component of the software reliability model implies that the assignment of failures to failure correc-

tion personnel will be essentially random in time, since no section of code (and the programmer responsible for it) is being favored in the testing process. Thus each debugger may be viewed as a Poisson server in a queueing system with an input of identified failures. Experience has indicated that testing is stopped whenever an excessive backlog\* of failures is building up for any one debugger and hence impeding the identification and correction of other failures in his program area. The value of the failure correction personnel utilization factor  $\rho_F$  may therefore be determined, by limiting the queue length  $m_Q$  (at a given probability level  $P_{m_Q}$ ), as

$$\rho_F = \left(1 - P_{m_Q}^{1/P_F}\right)^{1/m_Q}. \tag{10}$$

Using the assumptions that have been made, calendar time intervals can be related to execution time intervals, and employing (5), to MTTFs. [12] The calendar time interval  $\Delta t$  is given by

$$\Delta t = M_0 T_0 \sum_{k} \frac{1}{P_k \rho_k}$$

$$\left[ \mu_k \left[ \frac{1}{T_{k_1}} - \frac{1}{T_{k_2}} \right] + \frac{\theta_k}{C} \ln \left[ \frac{T_{k_1}}{T_{k_2}} \right] \right]. \tag{11}$$

The index k can have the values C. F, or I, corresponding to the computer-limited, failure-correction-personnel-limited, or failure-identification-personnel-limited periods, respectively. The quantities  $T_{k_1}$  and  $T_{k_2}$  represent the MTTFs at the boundaries of these periods, P is the resource quantity, and  $\rho$  is the resource utilization. Ordinarily,  $\theta_F = 0$  and  $\rho_I = 1$ .

The boundaries of the different resourcelimited periods are the present and objective MTTFs and the transition points

$$T_{kk'} = \frac{C\left(P_k \mu_k' \rho_k - P_k \mu_k \rho_k'\right)}{P_k \rho_k' \theta_k - P_k \rho_k \theta_k'} \tag{12}$$

that lie within that range, where (k, k') have the values (C, F), (F, I), and (I, C).

Some further details are discussed in Appendix B.

# VI. ESTIMATION OF MODEL PARAMETERS

A number of parameters have been defined in preceding sections that must be evaluated in order to specify the reliability model completely. The essential ones are listed in Table I. The weighted average of parameter values used for four projects for which complete data is presently available is given where they have any possible significance for other projects. The parameters are grouped into four categories: planned, debug environment, test environment, and program. Predictions affected by the parameter are indicated.

The planned parameters  $P_C$ ,  $P_F$ , and  $P_I$  are determined by the computer and personnel resources available. The parameter  $P_C$  represents computer time in terms of prescribed work periods. For example, if available computer time per week is 75 hours and the prescribed work week is 37.5 hours, than  $P_C = 2$ .

The computer utilization factor  $\rho_C$  should be set at its actual value. Note that this factor is often either implicitly or explicitly controlled in order to maintain satisfactory turnaround time. The failure correction personnel utilization factor  $\rho_F$  is computed using (10). Experience thus far seems to indicate that controlling the failure queue length so that it does not equal or exceed 3 with probability 0.9 appears reasonable; these values may be refined with more knowledge of the effect of backed-up failures on the debugging process.

The MTTF objective  $T_F$  is generally set based on external considerations such as

<sup>&</sup>quot;In measuring backlog we do not count failures whose correction is deferred for reasons other than availability of personnel to fix them.

TABLE I
ESSENTIAL PARAMETERS OF RELIABILITY MODEL

Parameter	Definition	Category	Average Value Over Projects 1-4	Predictions Affected
Pc	available computer time (measured in terms of prescribed work periods)	planned	_	Δτ
PF	number of available failure correction personnel	planned		Δt
P <sub>1</sub>	number of available failure identification personnel	planned		Δι
PC	computer utilization factor	planned		· \Delta t
PF	failure correction personnel utilization factor	planned		Δι
T <sub>F</sub>	objective MTTF	planned	na contrate distribution	$\Delta m, \ \Delta \tau, \ \Delta t$
θς	average computer time expended per unit execution time	debug environment	1.68	Δι
θ,	average failure identification work expended per unit execution time	debug environment	2.86	Δι
#c	average computer time required per failure	debug environment	2.03 hr	$\Delta t$
#F	average failure correction work required per failure	debug environment	6.42 hr	Δt
μ1	average failure identification work required per failure	debug environment	2.01 hr	Δι
C	testing compression factor	test environment	12.5	$T$ , $\Delta m$ , $\Delta \tau$ , $\Delta t$
M <sub>0</sub>	number of failures required to expose and remove all errors	program	energinen (h. 180 <b>2).</b> Disebberari <b>1</b> 00	$T$ , $\Delta m$ , $\Delta \tau$ , $\Delta t$
<i>T</i> <sub>0</sub>	initial MTTF at start of testing	program	consider the second	$T$ , $\Delta m$ , $\Delta \tau$ , $\Delta t$

economics and the MTTFs of other system components.

The debug environment parameters should be obtained from data taken in the actual project debug environment or one that closely resembles it. By "environment" we are referring to factors such as batch debugging vs.

interactive debugging, debugging aids available, computer used, language used, administrative and documentation overhead associated with corrections, etc. The parameters may be evaluated by collecting data on failure identification and correction work and computer usage profiles (with execution time) and fitting the data with

$$x = \theta \tau + \mu M_0 \qquad (13)$$

$$\left[ 1 - \exp \left[ -\frac{C}{M_0 T_0} \tau \right] \right].$$

using a weighted least squares criterion.

It appears likely that some or all of the debug environment parameters will prove to be relatively constant over wide varieties of projects or classes of projects. As more data is collected, it should be possible to gain a deeper understanding of what ranges of values these parameters take on and what factors they are dependent on. This knowledge should make it possible to increase the accuracy with which the calendar time required to meet a mean time to failure objective can be predicted.

The testing compression factor C depends on the test environment and its relationship to the actual operating environment of the program. More specifically, its value is related to the extent to which redundancy due to runs being made under identical conditions during the operation of the program is removed during the test phase. There is some indication that C is reasonably stable across similar test environments, but data from more projects is needed to verify this. The quantity C may be computed for a project after a sufficient period of operation has occurred following the test phase so that operational phase initial MTTF  $T_0$  can be accurately estimated. By use of (5) and the fact that the MTTF at the end of test will be equal to  $T_0$ , we obtain

$$C = \frac{M_0 T_0}{\tau} \ln \left[ \frac{\bar{T}_0}{T_0} \right], \tag{14}$$

where  $T_0$  is the MTTF at the start of the test phase and  $\tau$  is the total execution time for the test phase. If there is no good way of estimating C for a particular project, it is probably best to be conservative and take C = 1.

The two program parameters must be initially estimated from characteristics of the program itself. However, once data is available on intervals between failures during test or operation, these parameters may be reestimated. The accuracy with which they are

known generally increases with the size of the sample of failures.

The parameter  $M_0$  must initially be estimated from the number of inherent failures  $N_0$  and the error reduction factor B, using (2). Data can be developed on average error rates (errors per instruction) at the start of various phases of testing; these data will permit  $N_0$  to be estimated. Data taken in this study and data taken by Akiyama [14] and Endres [15] give a range of 3.36 to 7.98 errors per thousand source instructions for assembly language programs at the start of system test; the weighted (by number of instructions) mean is 5.43 errors/K inst. For a detailed discussion of the determination of B, see Appendix D.

The parameter  $T_0$  must initially be estimated from

$$T_0 = \frac{1}{fKN_0} \,. \tag{15}$$

where f is the linear execution frequency of the program or the average instruction execution rate divided by the number of instructions in the program and K is an error exposure ratio which relates error exposure frequency to "error velocity." The error velocity is the rate at which errors in the program would pass by if the program were executed linearly. The error exposure ratio accounts for:

- (a) Code is not "straight line" but has many loops and branches, except in very trivial cases, and
- (b) The machine state varies, and hence the error associated with an instruction may or may not be exposed at one particular execution of the instruction.

At present, K must be determined from a similar program. It may be possible in the future to relate K to program structure in some way. On six projects for which data is available, K ranges from  $1.54 \times 10^{-7}$  to  $2.99 \times 10^{-7}$ . Therefore, intervals between failures typically represent cycling through the program a large number of times.

Some of the practical aspects of parameter estimation are discussed in Appendix D.

# VII. PROGRAM PARAMETER REESTIMATION

Maximum likelihood estimation is used to reestimate  $M_0$  and  $T_0$  as testing progresses. The important relationships are given in Appendix E. A portable FORTRAN program has been developed [16],[17] to perform the calculations, so a graphical view will be provided at this point.

The essential data required for the reestimation are the execution time intervals between failures experienced in testing. A set of such intervals have been plotted against the sequential numbers of failures in Fig. 2. The interval plotted for failure m represents the execution time between failure m-1 and failure m. The set of intervals constitutes a sample from a multivariate probability distribution.

The execution time model provides the basis for plotting a curve of anticipated mean failure interval vs. failure number. This curve is determined by two parameters, the initial MTTF  $T_0$  and the total expected failures  $M_0$ . Values of the parameters are chosen to maximize the likelihood of occurrence of the set of intervals that have been experienced. In a very

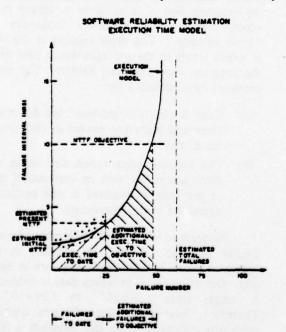


Fig. 2. Program parameter reestimation and project status monitoring.

approximate sense, the curve is being fit to the data indicated.

It will be seen from the nature of the curve that  $T_0$  can be estimated more accurately than  $M_0$ . The parameter  $T_0$  is the vertical axis intercept of the curve but  $M_0+1$  is the horizontal axis value that the curve approaches asymptotically. Note that the number of failures to date can be read from the figure. The execution time that has elapsed to date is the sum of the failure intervals experienced; it is approximated by the area under the curve.

Since reestimation of the parameters is a statistical process, one also wishes to determine confidence intervals. The curve should now be viewed as a band, with its thickness increasing with the magnitude of the confidence interval (a 90 percent confidence interval requires a thicker band than a 75 percent one). The confidence interval for  $T_0$  is given by the range of vertical axis intercepts. The confidence interval for  $M_0$  is given by the range of asymptotes approached.

# VIII. MONITORING TESTING PROGRESS

Since  $M_0$  and  $T_0$  are continually refined by reestimation during test, it is easy to continually reestimate:

- (a) Present MTTF  $T_P$ , using (5) with  $\tau$  set equal to execution time used to date.
- (b) Remaining execution time  $\Delta \tau$  required for the maximum likelihood estimate of T to reach an established MTTF objective  $T_F$ , using (8) with  $T_2 = T_F$  and  $T_1 = T_F$ .
- (c) Remaining calendar time  $\Delta t$  required to reach  $T_F$ , evaluating (11) over the interval  $[T_P, T_F]$ .
- (d) Remaining failures  $\Delta m$  required to be experienced to reach  $T_F$ , using (7) with  $T_1 = T_F$  and  $T_2 = T_F$ .

Some of these quantities may be viewed on Fig. 2. The present MTTF is established as the vertical axis value of the intersection of the curve with the vertical line that represents number of failures to date. The intersection of the MTTF objective line with the curve determines the number of additional failures required to meet the objective (increment

measured along horizontal axis) and the additional execution time required (area increment under the curve).

Confidence limits may be established (in addition to the "most likely" values) by carrying out the computations for corresponding confidence limits of  $M_0$  and  $T_0$  (e.g., the confidence limits for  $\Delta \tau$  are the remaining execution times required for the confidence bounds of T to reach  $T_F$ ). The confidence limits may be seen pictorially on Fig. 2 by viewing the curve as a band.

The total set of results is useful to managers for planning schedules, estimating status and progress, and determining when to terminate testing. The 75 percent confidence level estimate was found from experience to be the most useful from the manager's viewpoint, although the 50 percent, 90 percent, and 95 percent confidence levels contribute some added insight.

The FORTRAN program previously mentioned also computes estimates of  $T_P$ ,  $\Delta m$ ,  $\Delta \tau$ , and  $\Delta t$  from the set of intervals between failures and certain required model parameters. A sample report output by this program for an actual project is shown in Fig. 3. Note that "999999" indicates "no upper limit."

It is thus possible for the manager to obtain on a regular and continual basis, during the system test phase of a project, information such as the following (75 percent confidence intervals will be stated):

- (a) At present the best estimate of total possible failures during the maintained life of the program is 142 with a confidence interval of 136 to 152.
- (b) The best estimate of MTTF prior to testing is 0.847 hours (confidence interval from 0.701 to 0.992 hours).
- (c) The present MTTF is 20.4 hours (confidence interval more than 12.5 hours). This represents 73.4 percent (confidence interval from 45.1 percent to 100 percent) of the MTTF objective of 27.8 hours.
- (d) In order to meet this objective, the best estimate of the additional number of failures to be experienced is 2 (confidence interval from 0 to 7).

- (e) It is estimated that meeting the objective will require 2.46 additional computer hours of testing (confidence interval from 0 to 7.94) and one working day (confidence interval from 0 to 4).
- (f) Expected completion date is 11/12/73 (confidence interval from 11/9/73 to 11/16/73).

It should be noted that the confidence intervals are of greatest significance to the manager, although the maximum likelihood estimate is of interest.

Although the estimation program has proven to be of most value during the system test phase, nothing prevents it from being applied to individual programmers who are testing subsystems. If the subsystems are too small, however, estimates that are made may have wide confidence intervals for a large part of the test period, and hence be of limited value. If calendar time predictions are to be accurate, you may wish to use resource parameter values that are either measured or adjusted to apply to the particular individuals involved.

The program can also be used during the operational phase of a project. If the software is being maintained (i.e., failures are being corrected provided the errors causing them can be found), the operational phase is handled just like a test phase except the testing compression factor C is set to 1. If failure intervals from a test phase and the operational phase are used together, the former intervals should be increased by a factor of C (and the parameter C should be set to 1 as before). If the software is no longer being maintained, software reliability estimation programs [16],[17] can still be employed. MTTFs with confidence intervals can be estimated, but  $M_0$  will generally be infinite (B is 0, allowing  $N_0$  to be finite) and the quantities  $\Delta m$ ,  $\Delta \tau$ , and  $\Delta t$  required to reach the MTTF objective cannot be determined.

Note that the program correctly estimates the situation with regard to failures at all times; determination of inherent errors  $N_0$  or additional net errors  $\Delta n$  that must be corrected to attain the MTTF objective requires knowledge of the error reduction factor B.

# SOFTWARE RELIABILITY PREDICTION PROJECT 1

BASED ON SAMPLE OF 136 TEST FAILURES EXECUTION TIME IS 25.34 HRS 27.80 HOURS CALENDAR TIME TO DATE IS 96 DAYS PRESENT DATE:11/9/73

		CONF.	LIMITS		MOST		CONF.	LIMITS	
	954	90%	75%	A008	LIKELY	50%	758	901	95%
TOTAL PAILURES	136	136	136	13	142	148	152	163	182
INITIAL HTTP(HR)	0.522	0.617	0.701	0.744	0.847	0.949	0.992	1.08	1.17
PRESENT MITT (HR)	999999	999999	999999	30.9	20.4	14.5	12.5	9.53	7.05
PERCENT OF OBJ	100.0	100.0	100.0	100.0	73.4	52.0	45.1	34.3	25.4
•••	ADDITIO	NAL REQU	UI REMEN	TS TO ME	ET MTT	OBJEC	TIVE ***		
PAILURES	0	0	0	0	2	5	2	12	23
EXEC. TIME (HR)	0	0	0	0	2.46	6.09	7.94	12.4	19.4
CAL. TIME (DAYS)	0	0	0	0	0.958	2.85	4.03	7.39	13.8
COMPLETION DATE	110973	110973	110973	110973	111273	111473	.111673	112173	112973

Fig. 3. Sample project status report as generated by FORTRAN program.

## IX. APPLICATION OF THEORY TO PROJECT SCHEDULING AND MONITORING

The software reliability theory described in this paper is currently being applied to the monitoring of a number of software development projects, as indicated in Table II. The projects are being conducted both inside and outside Bell Laboratories. They vary in size from 3 to about 300 people and about 5000 to over 2,000,000 instructions and include real-time control systems, analysis tools, interactive systems, military systems, business systems, and systems with large data bases.

The MTTFs predicted at the end of system test for the four projects that have completed their life cycles are compared with measured MTTFs in Table III. In all four cases, the measured value lies within the 50 percent confidence range of the prediction.

Two typical plots from Project 1 are shown in Figs. 4 and 5. The center curve is the maximum likelihood estimate and the outer

curves represent the bounds of the 75 percent confidence interval, in each case. Both figures represent running histories of predictions throughout the course of the project, as indicated by the dates on the horizontal axis. Fig. 4 is a plot of present MTTF; it is useful for indicating progress toward the MTTF objective (some managers may prefer the alternative plot of "percent of objective attained"). The upper confidence limit for this quantity can be noisy when only a few errors remain, since a slight shift in the number remaining can affect present MTTF drastically. It is very common for the upper confidence limit for present MTTF to cease to exist at the end of the test period of a project with very few errors remaining. Fig. 5 is a plot of the current estimate of project completion date. Note that for a good part of October, the 75 percent confidence interval does not enclose the completion date predicted at the end of system test. This is to be expected; about 1/4 of the time it will not. The 95 percent confidence interval does enclose the final predicted completion date almost all the time.

#### TABLE II

## APPLICATION OF SOFTWARE RELIABILITY THEORY TO SOFTWARE DEVELOPMENT PROJECTS

Status	Number of Projects
Complete data taken—system test and operational phases	4
Data available from system test phase, currently taking data on operational phase	1
Complete data taken—system test phase	3
Currently taking data—system test phase	1
Actively preparing to take data—system test phase	11
Strong interest in applying	6
Analyzing or planning to analyze data from past projects	3
	-
Total	29

Some projects pass through several versions with substantial design changes between them. Figs. 6 and 7 were taken from such a project. The effects of the changes are exaggerated here because of the compression of the horizontal axis necessary to accommodate almost 1-1/2 years of data. Fig. 6 is a plot of the estimated total failures; note the substantial increase that occurs at each design change (the change in the estimate is not instantaneous; there is a small time lag until the estimation algorithm realizes that the change has occurred). Also observe the sharp increase in size of the confidence interval and subsequent reduction as the estimation algorithm copes with the new uncertainty. Fig. 7 is a plot of present MTTF; note how MTTF drops when substantial design changes are made and slowly rises during periods of steady debugging. Further discussion of the handling of design changes is contained in Appendix C.

Studies are currently in progress to apply the reliability monitoring techniques to operating system software in computation centers. Computation center managers commonly have the problem of reconciling demands for new software features and improvements with requirements for reliability of service. Adding new features usually results in a period of reliability degradation, until the new features are thoroughly exercised and errors corrected. Decisions as to when to add new features have previously been based on intuition. The new reliability monitoring techniques (using present MTTF) should make it possible to set low and high service objectives and control addition of new features to keep MTTF approximately between these bounds (new features would be added when the high objective is exceeded and a freeze on changes would be imposed when MTTF drops below the low objective).

## X. APPLICATION OF THEORY TO SYSTEM ENGINEERING

The manner in which this software reliability theory has been developed results in a compatibility with hardware reliability theory that permits combination of hardware and software components in determining overall reliability of complete systems. The techniques of reliability budgeting or allocation commonly used in system engineering can therefore be applied to systems involving hardware and software components.

Two approaches to this combinatoric problem are presented. The first deals with components that function concurrently; it represents the conventional combinatoric approach used for hardware systems. The second approach relates to components that function sequentially. It was developed originally for pure software systems and is probably most applicable to them; however, it can also be applied to hardware or mixed systems if the hardware components satisfy the assumed conditions.

A system is considered to be composed of concurrently-functioning components if the satisfactory operation of the system is dependent on continuous satisfactory operation of each com-

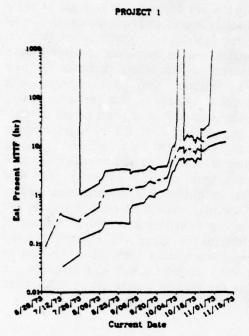


Fig. 4. Present MTTF history for Project 1.

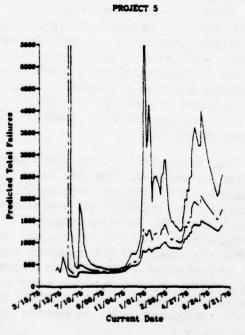


Fig. 6. Predicted total failures history for Project 5.

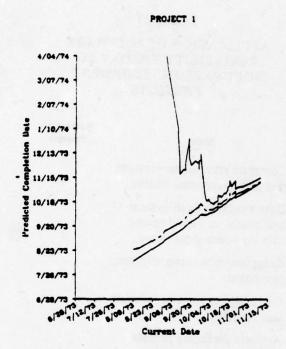


Fig. 5. Predicted completion date history for Project 1.

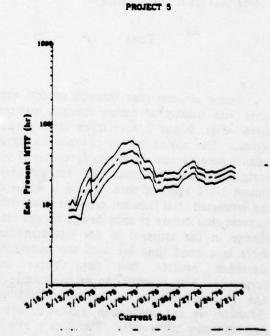


Fig. 7. Present MTTF history for Project 5.

TABLE III
COMPARISON OF MEASURED AND PREDICTED MTTF (HOURS)

	Project 1	Project 2	Project 3	Project 4
Measured (use period)	14.6	31.4	30.3	9.2
Predicted (at end of test)				
Max. Likelihood Value	20.4	43.5	30.4	14.5
50% Confidence Range	14.5-30.9	>24.5	>16.0	7.6-27.9
75% Confidence Range	>12.5	>19.8	>11.9	5.7-39.4
90% Confidence Range	>9.5	>12.1	>5.8	>2.8
95% Confidence Range	>7.1	>5.8	>0.8	>0.7

ponent or set of components (or alternatively, if failure of any component or component set at any time can affect system operation). "Component set" is mentioned so that the case of multiple redundant components is included, where the entire set must fail before system operation is affected.

The foregoing type of system is analyzed by drawing a "failure logic" or "successful operation logic" diagram of the system and applying the following combinatoric rules to the AND or OR combinations indicated:

(a) If all k components must function successfully for system success (AND condition), or the failure of any of the k components will cause system failure, the system reliability R is given by

$$R = \prod_{i=1}^k R_i \ . \tag{16}$$

where R, are the component reliabilities.

(b) If successful functioning of any of k components will result in system success (OR condition), or the failure of all k components is required for system failure, the system reliability is given by

$$R = 1 - \prod_{i=1}^{k} (1 - R_i)$$
 (17)

This formula is derived from the principle that the overall failure probability is equal to the product of the individual failure probabilities.

If the component characteristics are specified in terms of MTTFs, reliabilities are determined (assuming exponential failure distributions) from (6).

The fact that software components are involved requires that some special factors be considered. One must be certain that all the "times" are either equivalent or reduced to an equivalent base. For example, hardware MTTFs are generally stated in actual calendar time, while software MTTFs are usually stated in execution (CPU) time. If two hours of calendar time elapse on the average for every hour of execution time (i.e., processor utilization is 0.5), the two different time measures must be referred to a common base. Note that if the system being considered is divided into several software components, the adjustment factors will change, since the utilization factors of each of the components must be used.

On the other hand, software MTTFs are sometimes stated in calendar time. Now, no adjustments are required. However, the MTTF of a software component will no longer be constant when the component is moved to a different system, even if the processor speed is

the same. The MTTF is then adjusted by multipying by the ratio of utilizations. It is probably best to measure component MTTFs in terms of execution time. They can then be adjusted to calendar time to fit any system utilization.

Consider a simple example. Fig. 8 represents a time-sharing system running on a multiprocessor machine. The time-sharing system runs under a general-purpose operating system. The reliabilities of the hardware for a 10-hour prime shift are labeled. The operating and time sharing systems have MTTFs measured in execution time of 20 hours and 40 hours, respectively. Each runs about 20 percent of the time and the other 60 percent of the processor capacity is used by application programs. We wish to find the prime shift reliability of the entire system.

The calendar time MTTFs will be 100 hours and 200 hours, respectively, and the prime shift reliabilities are found using (6)

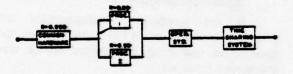


Fig. 8. Failure logic diagram.

as 0.905 and 0.951. The combined reliability of the two processors is found from (17) as 0.9999. Combining all the "series" elements by use of (16), we find an overall system prime shift reliability of 0.856.

A system is considered to be composed of sequentially-functioning components if only one component functions at a time and the satisfactory operation of the system is dependent on the satisfactory functioning of each component when it is active. This situation is particularly characteristic of a system composed of a number of modules, each of which has control at a given time. Littlewood [19] has developed a model for such systems. The model assumes that the system consists of k components

among which control is switched randomly according to a semi-Markov process; i.e., the transition probabilities between components are dependent only on the identity of the preceding component. The probability distributions of the sojourn times of the components in the active state are not restricted in any way. Failures for the i-th component are assumed to occur in accordance with a Poisson process with rate  $\lambda$ ,. If the  $\lambda$ , are small, then the system process is Poisson with rate  $\lambda$  given by

$$\lambda = \frac{\sum_{i=1}^{k} \sum_{j=1}^{k} \rho_{i} p_{ij} \mu_{ij} \lambda_{i}}{\sum_{i=1}^{k} \sum_{j=1}^{k} \rho_{i} p_{ij} \mu_{ij}}.$$
 (18)

The  $\rho_i$  are the equilibrium probabilities given by

$$\rho_i = \sum_{j=1}^k \rho_j p_{ji} . \tag{19}$$

The transition probability from component i to component j is given by  $p_{ij}$ , and  $\mu_{ij}$  is the mean duration spent in component i before switching to component j.

The rate  $\lambda_i$  is the reciprocal of the MTTF  $T_i$  [12, Eq. (7)], and hence it is readily derived from reliability by the use of (6).

It should be noted that combination of large numbers of small modules is generally not practical if knowledge of  $T_i$  for the modules must be gained by estimation, due to the problems of limited failure sample size.

## XI. CONCLUSION

The theory outlined in this paper has proved to be a good framework for understanding, measuring, and predicting the failure process in computer programs in terms of reliability and mean time to failure. Furthermore, it constitutes an approach that is compatible with hardware reliability theory as to combination of components and thus permits reliability analysis of hardware-software systems. The theory has been applied to several software development projects of different kinds and substantial

experience has been gained in its use. It is currently being employed or about to be employed on a larger set of projects. Studies now in progress indicate that the theory also holds considerable promise of usefulness in monitoring computation center service reliability and employing this knowledge in controlling the timing of software changes.

#### APPENDIX A

### GLOSSARY

- computer time: the measure used for allocating computer resources. It is often the residence time of the program in the machine (perhaps divided by the number of programs that can be concurrently resident in the case of multiprogramming).
- continously global: a property of a test effort; a test effort is continuously global if essentially all code is exercised in each of a set of test intervals, with lengths of perhaps a day apiece.
- error: the software defect that causes a failure.
- execution time: operation time of the central processor(s).
- failure: an unacceptable departure of program operation from program requirements.
- failure correction: the process of determining the error that caused a failure, removing the error, and proving that the failure no longer occurs.
- failure identification: the process of comparing test results against program requirements to establish the failures that have occurred.
- software reliability: probability of failure-free operation in a specified environment for a specified time.

#### APPENDIX B

## SOME DETAILS RELATING TO EXECUTION TIME AND CALENDAR TIME MODELS

The independence of one program failure from another postulated in Assumption 2 is met by most programs. There are very many sources and reasons for errors, involving different persons and different program functions. This very diversity would indicate independence. However, even if the same error source is causing two or more errors (e.g., a typographical error in a variable name in two different places), it is unlikely that the failures resulting from these errors have much interdependence (i.e., one causes the other).

One might think that the constant average failure occurrence rate mentioned in Assumption 2 implies that all programmers code equally well. It does not. The implication would be necessary only if the code executed between each pair of failures were developed by a single and different designer. Ordinarily, the time between failures is sufficiently large and the program structure is sufficiently complex that the executed code will have been associated with several programmers. Furthermore, these programmers can usually be viewed as constituting a random sample from the population of project programmers. Therefore, the concept of a constant average failure rate does not depend on identical error rates of individual designers.

The model is not dependent on the nature of the errors; they may result from single or multi-instruction defects, from wrong code, missing code, superfluous code, or misplaced code. Neither is it dependent on the cause of the errors such as requirements misunderstanding, poor interface definition or communication, poor algorithm, "stupid" error, etc.

It was noted that  $M_0$  represents the total number of failures possible during the maintained life of the software. Whether or not this number is reached depends on the extent to which the software is operated. After the end of the maintained life of a program, the number of remaining errors will remain constant, the possible number of failures will be unbounded, and the MTTF will be constant.

The resource expenditure parameters of the calendar time model represent average values. It is assumed that the project for which calendar time predictions are being made is large enough so that small-sample effects of different programmer skill levels and levels of task difficulty may be ignored (unless they are known and can be accounted for). Reasonable results have been obtained on projects as small as 5 programmers and 5000 instructions.

It may further illuminate the concept of the utilization factor to consider how use of selective overtime for those debuggers who have a failure backlog affects the calendar time required for the testing effort. Let  $\alpha$  be the overtime fraction (ratio of overtime to standard time for each work day). Now the effective available personnel are increased by the factor  $(1 + \alpha)$  and hence the calendar time for the failure-correction-personnel-limited segment of the test phase is reduced by this factor.

The probability for each debugger that there are one or more failures awaiting correction or being corrected is  $\rho_F$  [12]. Hence the actual work required is increased by the factor  $(1 + \rho_F \alpha)$ . For example, if  $\rho_F = 0.2$  and  $\alpha = 0.2$ , and we assume that the entire test phase is failure-correction-personnel limited, then a 20 percent improvement in calendar time is obtained at the expense of 4 percent in additional work. The value of selective overtime is clearly demonstrated.

# APPENDIX C HANDLING DESIGN CHANGES

The theory developed in this paper did not originally account for design changes that might occur during test, causing the total errors and therefore the total possible number of failures to increase. One could develop a more complex model than the piecewise exponential to handle error growth (e.g., a Wiebull model) but this would result in a loss of intuitive and analytical simplicity. Furthermore, there would be more parameters to be estimated, probably with less accuracy possible for any given sample size. Finally, the error growth process is not well known, hence the greater possible precision of a more complex model does not seem warranted. Three approaches for dealing with error growth, based on the piecewise exponential model, will be described.

The first approach is best suited to the situation where there are a number of small changes introduced continually throughout the test period. In this case, our basic hypothesis of a constant average error occurrence rate is approximated satisfactorily as long as:

- the changes are well distributed across the execution space of the program at all times.
- (b) the size of the changes in terms of number of instructions has a statistical distribution that is time-invariant, and
- (c) the code in the changes on the average is debugged to the same degree as the original code at the start of system test.

Experience to date has indicated that these assumptions are usually satisfied:

- (a) Distribution of changes may be correlated with program function and therefore, be nonrandom in memory space; however, it is usually not localized in execution space unless the function in which they are localized also happens to be highly periodic.
- (b) No compelling reason for assuming time-dependence of change size could be demonstrated; analogous data on effort required to correct errors [13] shows no time-dependence.
- (c) Although code in any particular change may be better or more poorly debugged than the original code, no reason could be found to justify a consistent difference nor does experience indicate such a difference.

From a large number of simulations it has been determined that maximum likelihood predictions of  $M_0$ ,  $T_0$ , and other quantities during test will reflect the added failures introduced by design changes quite accurately as long as the ratio  $\Delta M_0/M_0$  is "small," where  $\Delta M_0$  is the number of failures added at one time. "Small" means that  $\Delta M_0/M_0$  is about 0.25 or less in the case where the changes are modifications to the original code and about 0.5 or less where the changes represent additions to the code (resulting in corresponding increases to the total size of the program).

The total failures  $M_0$  can be related to the inherent errors at the start of test No (and failures experienced m to errors corrected n) by appropriately reducing the value of the error reduction factor B, if one can assume that the rate at which errors caused by design changes are introduced is proportional to the program error detection rate. This assumption implies [12, Eq. (2) and (12)] that the rate of errors introduced by design changes must be proportional to the number of errors remaining. In many cases, this is a reasonable model of reality, because it implies that the rate of change decreases as testing proceeds. If one views design changes as corrections for requirements errors, then the above model is followed if the ratio of requirements errors identified to program errors identified remains constant throughout the test period. See Appendix D for further discussion.

The second approach applies to the case of a number of small changes well distributed across the program but introduced at one time; e.g., with a new release of a program. In this case, there is a "jump" increase  $\Delta N_0$  in the total errors (and a corresponding increase of  $\Delta M_0$  in the total possible failures). If the ratio  $\Delta M_0/M_0$  is small, the reestimation of  $M_0$ ,  $T_0$ , and other quantities during test will reflect the increases or decreases fairly realistically, as previously indicated.

When  $\Delta M_0/M_0$  is large, overshoots (i.e., overestimates which damp out with time) tend to occur which increase in severity with  $\Delta M_0/M_0$ . The overshoot tends to be greater for modified code than for added code. It tends to be worse for  $T_0$  than  $M_0$  for intermediate values of  $\Delta M_0/M_0$ , but the effect for  $M_0$  ultimately predominates at higher values. The effect on  $M_0$  tends to increase as m (the number of failures experienced to date and used in the estimation process) decreases; the effect on  $T_0$  tends to increase as m increases.

A sensible approach for dealing with overshoots seems to be to focus on the lower confidence limit, assuming that it more truly represents the state of the project at this point than the maximum likelihood estimate.

The third approach should be used when an entire subsystem or subsystems are added, where the subsystems are functionally separated from the rest of the program so that the new failures introduced are essentially localized (i.e., they are not well distributed across the entire system). In this case, the system should be divided into two or more separate subsystems. Work with each separately, and then combine reliabilities in accordance with reliability combinatorial rules. One example of this situation is an application program that is added to a well-debugged operating system. In fact, the operating system may be sufficiently reliable that any failures it would contribute to the overall system could be completely ignored.

It should be noted that the errors in each subsystem will probably be distributed with different average occurrence rates. Hence it is necessary to continually deal with each subsystem separately, even if they are viewed as being integrated at some point. Execution times and failures must be allocated to each subsystem. Note that to allocate execution times, one must know the average percentage of time each subsystem is executing. For this allocation to be valid, the intervals for which each subsystem runs must be small compared with the failure intervals so that average execution percentages can be validly applied (i.e., so that the Central Limit Theorem applies).

The quantities  $M_0$ ,  $T_0$ ,  $T_P$ ,  $\Delta m$ ,  $\Delta \tau$ , and  $\Delta t$  are estimated separately for each subsystem. For the estimation process, the mean time to failure objective of the system must be allocated among the subsystems by allocating reliability objectives. The estimation of  $\Delta t$  will require an allocation of resources (failure identification and correction personnel and computer time) among subsystems. To determine overall system figures, the  $M_0$ 's,  $\Delta \hat{m}$ 's, and  $\Delta \hat{\tau}$ 's are added, the  $T_0$ 's and  $T_p$ 's are combined using standard combinatorial rules on their associated reliabilities (the rules applied depend on the structure that relates subsystem failures to overall system failure), and the maximum of the  $\Delta t$ 's is taken.

The approach can theoretically be extended to any number of subsystems. However, the quality of statistical estimation may suffer at some point if the subsystems become too small to yield good samples of failures.

In discussing the first two approaches, the phrase "distribution of changes across the pro-

gram has been used." In determining which approach to employ, note that the distribution is in execution space rather than instruction space. For example, the changes may all be bunched in one routine but if

- (a) the average failure interval is large enough so that the code executed during that interval consists of a large number of different routines (of which the changed routine is one) executed a number of times each, and
- (b) only one or two newly added errors in a particular execution of the changed routine are activated by the particular state the system is in at that time (usually true).

it will be seen that the changes and the errors they introduce are well distributed in execution space.

It should be noted that when a change involves added code, the linear execution frequency f will decrease. This must be accounted for if one is computing the error exposure ratio (ratio of error exposure frequency to error velocity) K from  $T_0$ .

## APPENDIX D

# PRACTICAL ASPECTS OF PARAMETER ESTIMATION

The determination of the amount of available computer time must take into account anticipated preventive and corrective maintenance times.

The prescribed work period represents "available time" or the total amount of time that the average member of the project can work upon request. In general, available time is established by company policy or the practical consideration of the maximum time one can reasonably request a person to work. Normally, any one person will not work at the limit for an extended period, but it can happen. "Available" also implies reasonably short notice. It does not imply that the project member must be standing by for the available time, but it does imply, for example, that he or she must be able to work two hours overtime sometime prior to the next day, when requested at 4 PM.

The best estimate of the number of failure correction personnel  $P_F$  is given by the number of programmers available who were occupied full time in coding the various new or extensively modified units of the system under test.

The parameter  $P_I$  represents the number of members of the test team, or the number of personnel familiar with system requirements and test plans and available for executing tests and verifying proper operation of the system or identifying departures from proper operation.

The key concept in resolving questions of interpretation in counting resource quantities or requirements under special conditions that may exist on different projects is the idea of a resource being limiting. For example, support staff (programming librarians, technical writers, etc.) are not counted because they do not directly constitute a limiting resource. Of course, the amount of support available may affect the environment in which the project operates, so that the resource parameters  $(\mu_C, \mu_F, \mu_I, \theta_C, \theta_I)$  may be a function of the support level. That is, a higher level of support may reduce the amount of work done by the failure identification and failure correction personnel for each failure or per unit execution time.

The failure identification activity usually includes the generation of trouble reports because this represents a task that is "inline" with respect to identifying a failure and thus limiting. Tasks associated with test planning and test case development are usually (but not always) completed prior to the test effort and not included. The failure correction activity includes the task of program change documentation, if required to be completed during test rather than after.

It is assumed that each person involved in failure identification can run and examine the results from any test that is made. Failure correction peronnel are assumed to work only on failures that have been attributed to their respective design areas. If a person has sufficient background to work at either failure identification or failure correction, depending on where staff is required, and is permitted to do so, that person should be included in both counts. This does not represent duplication

because only one resource at a time can be limiting. The theory and programs based on it view the personnel numbers as available personnel; a person is not "assigned" simultaneously to two functions.

On some projects, separate failure identification and failure correction personnel are not used; i.e., each person does both jobs. Since these resources are essentially merged, one must merge them and the resource requirements as far as the calendar time model is concerned. The best way to do this is to first set  $\mu_F = 0$ ; this will cause the CF and FI transition points in (12) between different periods of resource limitation to occur at negative values of MTTF (i.e., be nonexistent) and the integrand for the failure-correction-personnellimited period to be zero. Thus there will be at most one transition point, IC, and failure correction work required cannot by itself be limiting. The resource requirement  $\mu_i$  should be set to a  $\mu_i + \mu_F$ . The value of  $P_i$  should be taken as the total number of available programming personnel. The values set for  $P_F$  and  $\rho_F$ will be immaterial since there can be no failure-correction-personnel limited period. Do not, however, set either one to zero, because this will lead to undefined results for the transition points and the integrand indicated above.

A relatively painless approach to collection of the data for the estimation of the debug environment parameters  $\mu_F$ ,  $\mu_I$ , and  $\theta_I$  is to obtain weekly resource expenditure totals from each person involved in testing. Note that time spent in proving suspected failures false is included and thus will be apportioned among real failures. In cases where only one person is identifying the failures, the data gathered will be very individual-dependent unless we average data for several "one identifier" projects involving different persons. An appropriate weighting function for the data in making the least squares fit (13) is probably the square root of the execution time associated with each data point, except for the failure correction parameter fit, which should use the square root of the number of failures actually corrected. Overhead such as vacations and absences, training and administrative activities should be included. Usually measurements made of the parameters will represent net times; they must be increased

by a suitable overhead factor. A typical value is 1.35.

It should be noted that poor computer turnaround time can affect the pace at which failures are corrected. If the total nonproductive wait time involved in correcting each failure is substantial in comparison with average failure work time required per failure then  $\mu_F$  should be increased by the ratio of failure correction work time plus wait time to failure correction work time. Note that there is usually only one approach to the machine for each failure identification; hence turnaround time has little effect on that process.\*

If the errors causing some of the failures cannot be determined, the  $\mu_C$  and  $\mu_F$  parameters that would ordinarily be employed must be adjusted by multiplying by the detectability ratio D (see below) to account for the reduced amount of resources required.

Although the error reduction factor B is not an essential parameter of the software reliability model, knowledge of it is necessary if one wishes to relate failures and errors. It may be computed from

$$B = D(1 + A)(1 - G)$$
. (D1)

The parameter D is the detectability ratio or proportion of failures whose errors can be found. It is usually I on development projects but often less than that in computation center operation. The quantity A is the associability ratio, the ratio of errors discovered by reading during test to errors discovered by testing proper. Discovery of errors during test by reading is usually the result of associating an error discovered by testing with closely related errors. The parameter G is the error growth ratio or the increase in errors per error corrected. This factor can account for errors spawned in correcting other errors and for errors added by design changes (see Appendix C). When error growth occurs,  $N_0$  must be interpreted as the initial inherent errors at the start of test.

<sup>\*</sup>Shooman and Bolsky [13] have collected data that indicate an average of 0.61 runs per failure for failure identification and 1.35 runs per failure for failure correction.

At present, the values of A,D, and G must be determined by taking data for the particular project or using values from a similar project. Probably some or all of these parameters will eventually either be found to be constant or to be determinable functions of the test or operating environment. Data taken by Miyamoto [4, p. 199] and his associates, interpreted in terms of the author's formulation of G for spawned errors only, would indicate values of G from 0.05 to 0.09 over a number of cases of development of general purpose software systems. This compare with values from 0 to 0.06 over the four projects reported in this paper.

#### APPENDIX E

# PROGRAM PARAMETER REESTIMATION DETAILS

The important relationships in the maximum likelihood estimation process are given below. For derivations, see [12, Appendix B].

We find  $\hat{M}_0$  implicitly from

$$\hat{\phi} = \phi \left[ \hat{M}_0, \ m \right], \tag{E1}$$

where o is the failure moment statistic

$$\hat{\phi} = \frac{1}{m\tau_{-}} \sum_{i=1}^{m} (i-1)\tau_{i}^{'},$$
 (E2)

in which the  $\tau_i$  are the execution time intervals between failures and  $\tau_m$  is the total execution time. Now

$$\phi \left[ \hat{M}_0, \ m \right] = \frac{\hat{M}_0}{m} - \frac{1}{\Delta \hat{m}}, \tag{E3}$$

where

$$\Delta \hat{\phi} = \psi \left[ \hat{M}_0 + 1 \right] - \psi \left[ \hat{M}_0 + 1 - m \right] \tag{E4}$$

and + is the psi (digamma) function [18].

Now  $\hat{T}_0$  is given by

$$\hat{T}_0 = C \bar{\tau}_m \left[ 1 - \frac{m}{\hat{M}_0} \hat{\phi} \right], \qquad (E5)$$

where  $\bar{\tau}_m$  is the sample mean of time to failure during test.

The variance of  $\hat{\phi}$  is given by

$$var(\hat{\phi}) = \frac{1}{(\Delta \hat{\psi})^2} \left[ \frac{\Delta \hat{\psi}'}{(\Delta \hat{\psi})^2} + \frac{1}{m} \right], \quad (E6)$$

where

$$\Delta \tilde{\psi}' = \psi'(\hat{M}_0 + 1) - \psi'(\hat{M}_0 + 1 - m) \quad (E7)$$

and  $\psi'$  is the trigamma function [18, p. 44]. Confidence intervals may be established for  $M_0$  by determining the values of  $M_0$  that correspond to values of  $\phi$  chosen so that

$$\phi = \hat{\phi} \pm \left[\frac{1}{1-P}\right]^{1/2} S.D. (\hat{\phi}) .$$
 (E8)

where

$$S.D.(\hat{\phi}) = [var(\hat{\phi})]^{1/2}$$
 (E9)

and P is the confidence level. The intervals are based on the application of Chebyshev's inequality. Although this inequality tends to produce conservatively large intervals, it is probably best to employ it until more experience has been gained with the reliability model of this paper.

The variance of  $\hat{T}_0$  is given by

$$var(\hat{T}_0) = \frac{T_0^2}{m}$$
. (E10)

Confidence intervals for  $T_0$  are found by applying (E8) with  $\hat{\phi}$  replaced by  $\hat{T}_0$ .

Experience indicates that the quality of estimation can be considerably improved for small sample sizes by smoothing  $\hat{\phi}$ . On the other hand, the smoothing should eventually be

removed because it impairs the responsiveness of the estimation algorithms as the number of remaining errors becomes small. After trying a number of different smoothers, a variable length smoother which builds up to a length of 40 samples at m=40 and then drops to a length of one sample (i.e., no smoothing) at m=79 was adopted. The smoother weights each value of  $\phi$  by the corresponding  $m^{1/2}$ .

It has been found convenient to take run time data in a log (the logging could be automated). A log could also be used for recording (possibly interactively) occurrence which might be a failure. Alternatively, trouble report forms could be used. If there are multiple testers, it is necessary to accurately record the time of day of each test and set up a procedure so that the test logs and trouble data may be conveniently interleaved in time sequence. Errors found by a programmer without test (by code reading or other means) during the test period should not be counted as failures.

Observation of failures may occur at the time of program operation or when analyzing program output afterwards. Therefore, it proved best to allow one or two days to elapse after a test run before using the failure interval data, so that faise failures could be weeded out and the results thoroughly analyzed to minimize missed failures. Experience indicates that, after analysis, one is more likely to miss failures than to report false ones. If a failure reoccurs before the error that caused it can be fixed (assuming that the error could be determined), then the repetition of the failure should not be counted.

Although our experience indicated that perhaps 95 percent or more of failures could be readily classified on examination as software or nonsoftware (hardware, operator, etc.), occasionally the choice became uncertain. The criterion that was followed was to classify the failure as nonsoftware if it could not be made to recur when the program was rerun with all software and with known nonsoftware inputs exactly the same.

Almost all failures could be readily associated with particular test times; those that could not, did relate to a fairly short time interval. A randomly selected value within the interval was

taken as the time of failure in those cases. A deliberate decision during the maintained life of the program that an error will not be fixed is equivalent to a redefinition of the requirements so that the former "failure" no longer exists as such and should no longer be counted.

If there are very few failures (e.g., 40 or less) during the test phase, the confidence intervals may be very broad due to the small sample size. Somewhat narrower intervals but a worst case prediction may be obtained by assuming that a failure occurs right at the end of testing.

If it is difficult to measure actual CPU time for the failure intervals, one can measure another quantity that is proportional to CPU time, on the average. One possibility is elapsed test time. As long as CPU utilization is approximately the same over different failure intervals, the results should be satisfactory. Note, however, that MTTF values are now specified in terms of the new quantity. This may be an advantage if the quantity is a natural one for the application. However, adjustment of the objective MTTF  $T_F$  may be necessary. The linear execution frequency f of the program with respect to the new quantity will probably be lower, as will the parameters  $\theta_C$  and  $\theta_I$ .

If the ratio of proportionality should differ for any reason for different parts of the system test period, failure intervals should be adjusted so they are all given in common terms. For example, if the quantity measured changes from  $I_1\tau$  to  $I_2\tau$ , then one can adjust to the initial type of measurement by adjusting the latter intervals by the ratio  $I_1/I_2$ . The original  $T_F$ , f,  $\theta_C$ , and  $\theta_I$  will not change.

If testing starts before integration is complete, then there may be periods of inactivity in testing and debugging due to waits for programs not yet ready. These periods must be estimated separately and added to estimates of remaining t or else t must not be viewed as "running" when they occur. Note that severe computer outages (one day or more) and delays due to requirements changes have not been included in the estimate of remaining test phase length and must be added if expected.

If it is desired to determine the status of a program that is released to the field with some

errors not fixed, back up in the test phase by the number of failures corresponding to the errors outstanding and run the prediction program at this point.

Experience has generally indicated that it is best to ignore severity in estimating the program parameters and testing progress quantities. It is better to develop MTTFs for each failure class by dividing the overall MTTF by the fraction of failures occurring in that class. This method takes advantage of the largest possible sample size and avoids the problem of allocation of MTTF objectives and resources to classes.

#### **ACKNOWLEDGMENT**

The author is indebted to R. E. Archer, Jr., D. T. Chai, P. A. Hamilton, J. Opacic, and H. A. Zablocki for their helpful comments and suggestions.

#### REFERENCES

- [1] J. D. Musa and F. N. Woomer, Jr., "SAFEGUARD Data-Processing System: Software Project Management," Bell System Technical Journal, SAFEGUARD Special Supplement, p. S247 (July 1975).
- [2] F. P. Brooks, Jr., The Mythical Man Month, Reading, Mass.: Addison-Wesley, 1975, Chapter 2.
- [3] Z. Jelinski and P. B. Moranda, "Software Reliability Research," in Statistical Computer Performance Evaluation, W. Freiberger, Ed. New York: Academic, 1972, pp. 465-484.
- [4] I. Miyamoto, "Software Reliability in Online Real Time Environment," in Proc. 1975 Int. Conf. Reliable Software, Los Angeles, Calif., Apr. 21-23, 1975, p. 198.

- [5] M. Shooman, "Probabilistic Models for Software Reliability Prediction," in Statistical Computer Performance Evaluation, W. Freiberger, Ed. New York: Academic, 1972, pp. 485-502; als in 1972 Int. Symp. Fault-Tolerant Computing, Newton, Mass., June 21, 1972, pp. 211-215.
- [6] \_\_\_\_\_\_, "Operational Testing and Software Reliability Estimation During Program Development," in 1973 IEEE Symp. Computer Software Reliability, New York, N. Y., Apr. 30-May 2, 1973, pp. 51-57.
- [7] N. F. Schneidewind, "An Approach to Software Reliability Prediction and Quality Control," in 1972 Fall Joint Comput. Conf., AFIPS Conf. Proc., vol. 41. Montvale, N. J.: AFIPS Press, pp. 837-847.
- [8] \_\_\_\_\_\_, "Methodology for Software Reliability Prediction and Quality Control," NTIS Rep. AD 754377.
- [9] \_\_\_\_\_, "Analysis of Error Processes in Computer Software," in *Proc. 1975 Int. Conf. Reliable Software*, Los Angeles, Calif., Apr. 21-23, 1975, pp. 337-346.
- [10] B. Littlewood and J. L. Verrall, "A Bayesian Reliability Growth Model for Computer Software," in 1973 IEEE Symp. Computer Software Reliability, New York, N. Y., Apr. 30-May 2, 1973, pp. 70-77.
- [11] \_\_\_\_\_\_, "A Bayesian Reliability Growth Model for Computer Software," J. Roy. Statist. Soc. (Series C, Applied Statistics), vol. 22, no. 3, pp. 332-346, 1973.
- [12] J. D. Musa, "A Theory of Software Reliability and Its Application," *IEEE Transactions on Software Engineering*, vol. SE-1, no. 3, pp. 312-327, Sept. 1975.
- [13] M. L. Shooman and M. I. Bolsky, "Types, Distribution, and Test and Correction Times for Programming Errors," in Proc. 1975 Int. Conf. Reliable Software, Los Angeles, Calif., Apr. 21-23, 1975, pp. 350-351.
- [14] F. Akiyama, "An Example of Software System Debugging," in Proc. 1971 IFIPS Conf., p. 359.

- [15] A. Endres, "An Analysis of Errors and their Causes in System Programs," in Proc. 1975 Int. Conf. Reliable Software. Los Angeles, Calif., Apr. 21-23, 1975, pp. 328-329.
- [16] J. D. Musa, Program for Software Reliability and System Test Schedule Estimation\_ User's Guide, available from IEEE Computer Society Repository.
- [17] J. D. Musa and P. A. Hamilton, Program for Software Reliability and System Test Schedule Estimation Program Documentation, available from IEEE Computer Society Repository.
- [18] H. T. Davis, The Summation of Series. San Antonio: Principia Press of Trinity Univ., 1962, p. 36.
- [19] B. Littlewood, "A Semi-Markov Model for Software Reliability with Failure Costs," in Proceedings of the MRI Symposium on Software Engineering, New York: April 20-22, 1976.

# ERROR COUNTING MODELS OF SOFTWARE RELIABILITY

Some Comments, Criticisms and Proposals

by

F.N. Parr

presented at

Software Life-cycle Management Workshop Airlie Virginia

August 1977

## 1. Introduction

A major problem in managing large-scale software projects has traditionally been the difficulty of measuring the quality of programs. Despite the wealth of published material advocating methodologies for improving software quality, their effectiveness has usually been judged subjectively rather than quantitatively. Reliability models which describe the frequency of failure of software system by a distribution whose parameters can be estimated empirically, provide one means of making these comparisons. The empirical measurement of program quality in the form of reliability is also an important ingredient for any attempt to explain the patterns which appear to govern the long-term development of large-scale software projects.

Since the reliability techniques of hardware systems had for some time been successfully used to improve designs and project management methods, it is not surprising that similar reliability models have been proposed for understanding the failures of software systems. Most of these models have been of the bug-counting type. They make little or no analysis of the internal structure of the software being considered. Future instantaneous failure rates are predicted on the basis of past failure history. The central idea behind such models is that a computer program will at any instant contain a fixed number of errors. As the program executes in its operational environment these bugs manifest themselves in the form of sporadic system failures whose frequency is taken to be proportional to the number of bugs remaining in the program. When a failure is observed efforts are made to remove its cause resulting in

removal of one error from the program after some fixed average number of failures in the case of models in (4) and (7), and in the case of the model in (3) a stochastically characterized jump in the number of errors remaining in the program. The effect of these models is, very roughly, to predict that the frequency of failure of software system decays in an approximately exponential fashion over time. This is the result of the error removal process proceeding at a rate proportional to the number of errors remaining. Using this decay law one can estimate the initial number of errors in the program from looking at its past performance and also predict its future reliability.

These reliability models have been tested on actual failure data from some operating systems and hardware system controllers giving predictions which were clearly of management value. However, there remain considerable problems in interpreting these results and even more so in extending the theory to deal with other types of software system not so directly concerned with real-time control. This paper explores the apparent limitations of the present methods and makes some suggestions for refining them.

Most of the problems with this approach to software reliability stem ultimately from the shallowness of the analogy between software and hardware systems. Hardware reliability theory is concerned with the period of time which elapses before an atomic system component becomes degraded and fails. The systems are repaired after failure by replacing the bad component with an identical (but working) copy. The lifetime of atomic components can be described probabilistically; sampling time-until-

failure from a large number of identical copies of the part will determine the distribution. The distribution of time-until-failure of a hardware system is in principle just a complex combinatorial calculation using the distributions of the lifetimes of its components. Contrast this with software systems where there is no degradation over time and where repair is not by replacement. There is no probabilistic model to describe the failure behaviour of atoms of software. Furthermore, the errors which cause unreliability in software are errors of design - a problem not usually tackled in the analysis of hardware reliability. Also hardware errors have a definite location whereas software errors can be an inconsistency between several sections of the code.

Further examination of the problems arising from this approach to software reliability is divided into three sections. Section 2 shows that a necessarily probabilistic model for the reliability of an essentially deterministic software system can be constructed only with careful consideration of its input space and usage. Section 3 concentrates on the special problems of analysing the reliability of programs which have complex interfaces with other software. Section 4 takes a life cycle approach and considers the effects of maintenance and system redesign on reliability.

# 2. The Input Space and Usage Considerations

The primary objective of a software reliability theory is to construct a model which describes probabilistically the occurrence of failures of a program so that the likelihood of any future pattern of failure behaviour can be predicted. Predictions of this nature are useful:

- (i) in managing the system which uses the software;
- (ii) in managing the repair and maintenance of the software itself; and
- (iii) in guaranteeing the quality of the software before it is actually put into live use.

Predictions for all three uses, but most particularly (iii), all depend on the assumption that there is no significant shift in the distribution of inputs presented to the system during the period of time under consideration. In analysing the reliability of hardware systems this assumption is easily justified. Although the lifetimes of some hardware components can be influenced by the extent to which the component has been used and stressed, the class of inputs having this effect is not precisely defined and unpredictable random effects still dominate the process. In contrast to this there is no reason in principle (although it may be extremely hard to do in practice) why the input space of a program cannot be partitioned into those for which it works and those for which it fails since its behaviour is repeatable whenever the input is exactly reproduced. So the reliability of a software system depends very directly on the inputs with which it is driven. If some probabilistic measure of software reliability

is to be given, then the only means by which stochastic features can enter the model is in the form of a probabilistic description of the process by which input data is selected. Now for certain types of systems this implicit assumption seems quite natural - for example when building a software controller for a hardware device whose operational characteristics are fixed and can be reasonably treated as stochastic fluctuations around some steady-state signal. However, moving away from the real-time embedded applications there are many other types of software for which this assumption seems quite inappropriate. To hope to obtain a reliability theory for such software or even to delineate those applications for which the bug-counting approach can hope to make useful predictions of failure rates, one must identify those qualities of the usage environment which cannot be characterized probabilistically and yet have a significant effect on the failure behaviour of the system. Just as performance analysis of computer systems has depended on careful examination and characterization of workload patterns, a complete software reliability model should include an analysis of the form of its workload.

In arguing for a software reliability theory which is more directly based on usage, one is forced to admit that even to specify in detail what constitutes the input space of a large-scale software system may be a profoundly daunting task. This is particularly true of real-time environments where sophisticated linguistic tools are required to describe all possible signal histories with which the system may be faced. But despite the difficulty of obtaining a complete formal specification of the allowable usage of a large system, most software developers recognize that

they do have some measure of control over the observed failure rates of their systems. At the most simple level it is standard practice to test new software with a sequence of input cases which are generally understood to be increasingly stressful. The rationale is to get the easy errors out first. Such an approach is a clear recognition of the fact that by adjusting the usage of software one can bias the mean frequency of failure observed.

One desirable advantage of MUSA's reliability analysis over the other bug-counting models can be viewed in this light. This is the suggestion that in measuring and predicting the intervals between program failures, total execution time should be used rather than elapsed actual time. This amounts to the observation that the intensity of testing and hence the observed failure rate of a software system can be adjusted very directly merely by performing tests at a different rate. An operating system which is run for only 12 hours per day or a suite of batch programs executed less often than usual can be expected to show a reduced rate of failure. Building this effect into a description of the error removal process gives rise to a more widely applicable model for reliability. A theory which attempted to predict the elapsed interval between failure of an application program which was run only occasionally at the whim of particular programmers would be grappling with the additional problem of modelling human actions.

Another way of altering the observed rate of failure is to specifically search for input cases which are particularly likely or unlikely to create problems for the software. This factor is specially relevant when

the behaviour of new software during development testing and debugging is used to predict its performance in operation. Most developing software contains errors which can actually be found by informal desk checking. Since no execution time with the actual system is consumed in detecting errors by this method, carrying this form of debugging would generate arbitrarily short intervals of execution time between each observed failure and correspondingly low reliability. Although purely analytical checking would be a completely impractical way for validating large-scale software there is a continuum of approaches lying between abstract correctness proving and simple-minded random selection of input data for test cases. In practice it is often possible to identify dangerous areas in a system. For example there may be some particularly dangerous interfaces or complex control flows which are likely to contain a higher density of errors than the rest of the code. Also there may be quite specific ways in which a program is suspected of behaving incorrectly (i.e., not as the designer intended) but which can be conveniently shown to be errors only by executing with particular test cases. These examples are concerned with achieving a speeded-up failure rate. Conversely in the early stages of system integration it is often helpful to select input data which avoids areas of difficulty in the program in order to achieve a partially working system as quickly as possible.

What is meant by a continuum of activities between formal correctness proving and testing with arbitrarily selected data can be made more precise by referring to symbolic execution systems () (). These systems interpret programs performing algebraic manipulations on expressions for

the data rather than evaluating variables as is done during normal execution. So for a program with no data dependent path decisions the output variables computed are expressed as formulae in the input variable values, enabling the program to be completely verified by one symbolic execution. Where there are data dependent path decisions to be made the program tester may choose whether to proceed theoretically, suggest loop invariants and try to obtain a complete correctness proof, or whether to instantiate state variable expressions with enough actual data to allow the interpretation to continue more as a test execution. By choosing between specific and general analysis independently at each decision point in the symbolic execution, the tester selects a level of abstraction for his validation from a wide choice. But this use of high-level predicates and inductive arguments in validating programs is just a formalization of the practice of thinking about the structure and semantics of the code in order to adjust the rate of extracting errors. The formalization offered by symbolic execution is useful because it suggests ways in which this aspect of the intensity of testing could be measured.

Besides this quite general testing concept, there are other ways in which the stressfulness of testing can be adjusted applicable to specific types of software. Scientific and numerical computation is an example requiring special treatment. Concepts such as stiffness of differential equations and poor conditioning of matrices are well established. The requirements on a suite of programs to process numerical data may very reasonably require that results are computed to a given accuracy or tolerance. The reliability estimate of such a suite would be quite meaningless unless

information were also kept describing how much poor conditioning there had been in the input stream which generated the operational data used in the model.

Moving away from mathematical software there are also fundamental choices to be made in selecting a test strategy for systems software which is intended to run in a wide (often potentially infinite) variety of configurations and installations. Typically this sort of software will contain a system generation facility which will put together an appropriate version of the system when given a description of an intended environment. In evaluating the quality of this software system some arbitrary balance is struck between testing the reliability of particular generated versions running in their environments and testing the reliability of system generation across all possible environments. These seem to give two entirely different concepts of what is meant by reliability and one would expect quite different and independent mean time between failure behaviour. Therefore, presumably by adjusting the testing strategy any intermediate value in a certain range of reliability could be demonstrated.

A final point concerns the variability of testing executions. .

Well-built computer software behaves deterministically in processing each set of input data. By repeating a successful (unsuccessful) test one can bias the observed reliability of the system upwards (downwards). Disregarding the remote possibility of this sort of measure being intentionally manipulated, one still has to face the fact that (after a certain stage in debugging) a program usually performs better on middle-of-the-road

standard input cases than in exceptional condition. Some rough measure of the extent to which test inputs and the execution environment have ranged over all possibilities is therefore needed. Past theoretical work in examining the completeness of various testing strategies has produced only limited results (5). One reason for this is that it has concentrated entirely on the characterization of tests by means of the control paths which they exercise. This is conceptually equivalent to making recommendations for the testing of totally unstructured programs composed of arbitrary control jump. But much of the debate in programming methodology of the last decade has resolved that unstructured programs are unmanageable and almost certainly untestable. It follows that progress in understanding the completeness of testing strategies will occur only when program structure is fully taken into account.

In summary we have identified a number of simple decisions about the usage of software which could be taken by management and which would influence the observed failure rate. Bug-counting based modelling of reliability should be carried out only in environments where:

- (i) the distribution of usage has been fixed once for all time in these respects; or
- (ii) the relative stressfulness of the different environments

  from which failure data has been collected can be

  estimated and taken into account.

This second condition is particularly relevant in predicting the installed behaviour of a system on the basis of its failure record during testing and debugging in an artificial software development environment.

# 3. Problems Arising from the Distributed Nature of Design Errors

In the preceding section it was observed that the primary function of a software reliability measure is to predict future program failure rates, and that in making such predictions account must be taken of possible differences in system usage between the testing period for which failure data was collected and the operational period for which failure rates are predicted. Since we do not at present know how to evaluate the relative stressfulness of different execution environments the obvious way of obtaining good reliability predictions for a program is to execute it with data as representative as possible of the expected execution environment. This can sometimes be achieved before installation by testing newly developed software live in its operational environment, but in many other cases such an approach would be impractical. For example the actual environment may not yet be built or there may be danger of the newly developed software severely damaging it. This section explores the problems of predicting the reliability of a software system with a bugcounting model on the basis of its performance in a simulated environment.

Although bug-counting models attempt to characterize reliability in terms of the number of errors remaining in a program, the input data on which they are based and the output phenomena predicted are both expressed in terms of system failure. This term therefore requires some scrutiny. Now it is usually assumed that there is some well-defined set of requirements which the software system is trying to satisfy and that it is contravention of these which constitutes failure. The problem is that these requirements are usually stated as loose assertions about the

behaviour of the complete system - software interacting with its environment. When the program being evaluated is an operating system or device controller them a failure can be defined as a crash or loss of response of the system. This is a relatively easy case since the failures can be roughly devided into those caused by hardware or operator malfunction as opposed to software errors. A rather more subtle characterization of failure would be required in estimating the reliability of a new software system which for example was required to interact with an existing data base. The problem is that for an interface between the old and new software of the complexity typically arising in practice no set of requirements on the new module to be developed could hope to completely characterize its input/output behaviour. Among the choices left open there are likely to be actions which alter the reliability of the interconnection of the systems.

Error counting analysis attempts to deal with the interconnection of two modules A and B in the following way:

errors  $E_{\mathbf{A}}$  are associated with module A errors  $E_{\mathbf{R}}$  are associated with module B

If testing and usage of A and B has been carried out independently using simulated environments then an estimate of  $E_A$  will be available and, assuming that B is the older (data base) module or system to which A is added, some estimate of  $E_B$  will also have been made. When the new software A is connected to the old system B two new sources of error will appear. Firstly, there will be the errors  $E_{AB}$  which are the result of interface mismatch. Secondly, there is the fact that as the new module provides

some new function to system B the usage of that system and hence its reliability must change. Conceptually this amounts to discovering errors which were in E<sub>B</sub> but were not discovered because of previous use of the system. Although a simulation of the environment of B could reasonably hope to find most of the errors in A, it seems unreasonable to hope that environment simulation would detect many of the other errors which will appear when the two components are integrated.

Besides the initial difficulty of predicting what will happen when modules are integrated, it is extremely difficult even after module A has gone live and been added to B to give any precise meaning to the reliability of A. System errors will presumably occur and can be detected. But analogous to the classification of operating system failures into hardware, software and operator generated there are now some very arbitrary decisions to be made as to whether the failure was caused by module A or not. Corrective action will often involve both modules. The requirements (used to define failures in the first place) of one or both modules may also have to be adjusted.

What emerges from this discussion is that it is extremely treacherous to try to define the reliability of a program which has to operate within a given software environment. In such cases the failures whose frequency is being modelled are not well defined events. The bug-counting approach is best suited to describing the behaviour of an isolated total system. For a total software system which is managed as a collection of components, the error-counting approach appears to give too little information about the locality of errors and the effects of interfaces to be useful.

## 4. Reliability of Programs Undergoing Maintenance and Development

Since there is no meaningful probabilistic description for the failure rate of small programs whose input space and functional requirements can be completely specified, software reliability theory is confined to describing the behaviour of large-scale systems. These large programs tend to be implemented, then developed and maintained over a considerable period of time. During this time their functional requirements will almost certainly be refined and may be significantly altered to meet new usage demands which were not foreseen by the original designers. Several authors have observed that these tendencies are quite general and can be formalized as laws which appear to govern the growth patterns of a wide class of software systems (2) (6). The complete history of a software project provides global (in time) information on program quality which is complementary to the view associated with error counting. Also the fact that software specifications do change over time presents some additional problems in failure interval analysis. It would therefore seem that some synthesis of the life cycle and reliability measurement approaches would be fruitful.

All published error-counting predictions of reliability have been smoothing models. That is to say that future instantaneous failure rates are estimated by averaging past failure rates, giving greatest weight to the most recent observations and extrapolating all data points along an exponential curve in time. This method is extremely conservative in the sense that it makes no provision for building in information about the development planned for the software. Hence the method is best suited to very short-term forecasting where the main effect is removal of errors.

For short intervals extrapolation of recent trends gives more definite projections than the plans for project development whose effect is bound to be difficult to quantify.

A common feature of very large or complex software is that it has to be developed incrementally. In other words, phases of testing and debugging alternate with the inclusion of new code expanding the capabilities of the system. This is done partly to ensure that the task of integration is carried out with as little of the system as possible at each stage and partly because some of the functional requirements of the system become apparent only during the development process.

The fact that systems for which reliability modelling is meaningful are usually developed in this way raises the question of how often estimates of the number of errors in a program should be reinitialized. In essence the proposed fitting of an exponential curve to the instantaneous failure rate is mathematically equivalent to extrapolating backwards from each observed failure and averaging the results to obtain an increasingly accurate estimate of the initial number of errors in the program at time  $\underline{t_0}$  say. When the program enters a new phase of development say at time  $\underline{t_1}$  new code will be added and presumably new errors injected with it. Subsequent failure data if extrapolated back to time  $\underline{t_0}$  gives an estimate not of the number of errors in the system at time  $\underline{t_0}$  but rather an estimate of that number of errors which if they had been present at  $\underline{t_0}$  and all the system had been available for testing in the interval  $(\underline{t_0}, \underline{t_1})$  would have left the actual number of errors in the enlarged system at time  $\underline{t_1}$ . Thus after injection of new code into the system, failure data from before that

estimating a different quantity. It so happens that the exponential law for error elimination has the effect of attaching a relatively heavy weight to the most recent observations and, therefore, prevents this bias from becoming overwhelming in the long term. However, it should be possible to obtain more accurate estimates by cutting out old data and reinitializing the model.

The above paragraph explains the importance of keeping track of developments to the system and using them in the reliability analysis. Up to this point reliability has been assumed quite specifically as mean failure rate. There is another aspect of software system quality which describes its ability to cope with unpredictable shocks - namely its ability to be developed and extended in ways unforeseen by the original designers. This quality is sometimes referred to as maintainability. When a software system is developed incrementally, its reliability will develop in sawtooth pattern with sharp rises in failure rate corresponding to the inclusion of new code and gradual declines in failure rate corresponding to the working out of errors during phases of testing and debugging. The life cycle approach to software would try to identify changes in the maintainability of the system by noting changes in the shape of each repeated section of the reliability curve.

Life cycle phenomena could in turn be used to refine the reliability model. For example the slowing of the rate of growth of software systems is usually attributed to their becoming less well structured as a result of taking on functions which were not part of the

original design. Loss of structure in a program makes errors markedly difficult to diagnose. One therefore expects the ratio of observed failures to errors removed to gradually increase as the system ages.

Another suggestion from life cycle analysis is that errors may propagate in an old poorly structured system. Correction of one error may create others. In (1) the chain reaction possibilities in this picture were invoked to explain dramatic changes in reliability between consecutive releases of an operating system. This possibility has obvious ramifications for error counting models.

### Conclusion

Bug-counting models of reliability provide some useful management information when used to describe the observed failure rate of an operating system working in one type of installation with a fairly static environment. To refine these models and widen their range of applicability requires:

- (i) identification of those qualities of the usage environment which can be altered and affect observed failure rates;
- (ii) better understanding of how to handle the reliability of interacting software components and interfaces; and
- (iii) the synthesis of life cycle and failure rate views in order to obtain explanation of how software reliability changes in the longer term.

### Bibliography

- L.A. BELADY, M.M. LEHMAN, "An Introduction to Growth Dynamics", in <u>Statistical Computer Performance Evaluation</u>, Academic Press, New York and London, 1972.
- M.M. LEHMAN, F.N. PARR, "Program Evolution and its Impact on Software Engineering", Proc. 2nd International Conference on Software Engineering, San Francisco, October 1976, pp 350 -357.
- 3. B. LITTLEWOOD, J.L. VERRAL, "A Cayesian Reliability Growth Model for Computer Software", 1973 IEEE Symposium on Computer Software Reliability, pp 70 77.
- 4. J.D. MUSA, "A Theory of Software Reliability and its Application", <u>IEEE Transactions on Software Engineering</u>, SE-1 #3 (September) 1975, pp 312 - 327.
- F.N. PARR, M.M. LEHMAN, "State of the Art Survey of Software Reliability", Report 77/15 Department of Computing and Control, Imperial College, London, 1977.
- L.H. PUTNAM, "Software Life Cycles", <u>Proc. 2nd International</u> <u>Conference on Software Engineering</u>, San Francisco, October 1976.
- 7. M.L. SHOOMAN, "Operational Testing and Software Reliability Estimation During Program Development:, Proc. of 1973 IEEE Symposium on Computer Software Reliability, pp 51 57.

STATISTICS DIVISION, MATHEMATICS DEPARTMENT, THE CITY UNIVERSITY, ST. JOHN STREET, LONDON ECLY 4PB

SOFTWARE RELIABILITY MEASUREMENT : SOME CRITICISMS AND SUGGESTIONS.

B. LITTLEWOOD, JULY 1977

## 1. Introduction

My intention in writing this paper is to provoke discussion about two aspects of software reliability measurement. The method I shall adopt is one of critical analysis of some previous research, together with tentative suggestions for future directions. In order not to everburden the argument, I shall not bend over backwards to praise the positive aspects of work I criticise: I am sure the authors concerned will be quite capable of performing this task, and in so doing help make for interesting discussion!

The first part of the paper will be concerned with the basic problems of definition. Before this provokes a cry of despair from the reader, I should emphasise that I see my task as being as much to convince him of the necessity of the operation, as selling my own favoured solutions. My general thesis will be that many of the reliability measures adopted for software rely far more upon hardware analogies, often unconsciously adopted, than on a careful analysis of the special requirements of software. Such an analysis, I believe, could reveal much research to be a mixture of technically sophisticated falsehood and meaningless nonsense. This, if true, is bad enough: but worse happens when these ideas are exposed to the innocent computer scientist having no background in reliability theory. shall quote from a recent paper in Computer [15], surveying methods of achieving reliable software, whose authors manage to be confusing about even the old hardware-derived reliability concepts, before they move on to the safer, deterministic ground of proving, structuring, testing, etc., which is their forte.

Even if satisfactory answers can be achieved to the problems in this area, it needs to be pointed out that we shall only be in a position (using the hardware analogy) to treat software as a black box (component). Let us not forget that the huge achievement of hardware reliability theory was the provision of methods of incorporating both stochastic information about component failure behaviour and deterministic information of the system's structure of components. Something similar is needed for software, particularly in view of the modern approaches of modularity and structured programming. My own belief, however, is that the hardware analogy will again be of little direct assistance. The second part of the paper will look at this question, and again I hope to be able to suggest areas in which effort should be concentrated.

## 2. Classical reliability measures

Let us begin by looking briefly at the measures which have been used in the hardware field. One of the most careful accounts is still that of Carlow and Proschan [1]. They give two basic measures which will generally have meaning: reliability and availability.

## 2.1 Reliability

Barlow and Proschan give two definitions. We shall consider here only the more general one :

Interval reliability is the probability that at a specified time, T, the system is operating and will continue to operate for an interval of duration x.

We shall denote interval reliability by R(x,T). Of course, in many cases we shall be interested in limiting interval reliability,

say R(x), given by

$$R(x) = \lim_{T\to\infty} R(x,T)$$
,

assuming that this limit does exist.

## 2.2 Availability

There are two definitions, both being in common use:

Pointwise availability is the probability that the system will be able to operate within the tolerances at a given instant of time, t.

We shall denote it by G(t).

Interval availability is the expected fraction of a given interval of time, (a,b), that the system will be able to operate within the tolerances (repair and/or replacement allowed).

We shall denote this by H(a,b). Barlow and Proschan define <u>limiting interval</u> availability to be

It is sometimes sensible to consider also a limiting version of G(t) as  $t \leftrightarrow \infty$ , we shall call this limiting pointwise availability.

# 2.3 How adequate are these?

Given that, for reasons of simplicity, it is necessary to summarise all available information into a single numerical measure of reliability, then one or other of these definitions will be appropriate for most situations. My own opinion, however, is that there is usually no such overriding need for simplicity and that more insight into the process of failures would be gained by considering, say percentiles of time-to-next-failure distributions.

Of course, striking a balance between simplicity and informativeness is a matter for fine judgment, but it does seem that in much writing on hardware reliability there is a preference for the simplists: rather than the simple. I am thinking, in particular, of the enormous popularity of mean-time-to-next-failure, mean-time-between-failures, and similar measures. Even in the case of hardware, it is regrettable that large amounts of failure data should still be summarised in such a crude fashion; for software, however, I hope to show later in this paper that use of such measures can be very misleading.

I have a technical criticism of one of the availability definitions which, again, will be particularly serious in the case of software. Consider interval availability: "the expected fraction of a given interval of time that the system will....operate....". What is really of interest is the actual fraction of time the system will operate, but of course, this is a random variable. Merely quoting the expected value of the random variable gives no idea of how much the actual result might deviate from this. We need the distribution of the fraction in order to be able to calculate a confidence interval.

In many practical situations, of course, it is the limiting form of the availability definition which is most appropriate. In such cases it might be thought that the fraction of an interval (0,T) that the system will operate would converge in probability, for large T, to the limiting expected fraction (limiting availability). This result would normally be established using Tchebycheff's inequality ([3], p.46). Unfortunately, convergence cannot be guaranteed. In fact, if we model the system's behaviour by an alternating renewal process ([4], p. 80 et seq.) with the two types of time interval representing operating and repairing, then it can be shown that the fraction of time spent working does converge in probability to

$$\frac{\mu_{W}}{\mu_{W} + \mu_{r}} ,$$

(where  $\mu_w$  and  $\mu_r$  are the means of the time-to-failure and time-to-repair distributions) as long as these means exist. If these distributions do not have moments, convergence may still take place, but it is possible to construct examples where it does not. If such a situation should be encountered, where the limiting interval reliability does not have the interpretation of (probabilistic) limiting fraction of time operating, it seems to me to be difficult to assign it any practical meaning. It is my contention that we are more likely to encounter this kind of difficulty with software than with hardware.

All of the above comments and criticisms have been essentially of the same nature: we should be extremely careful of replacing the wealth of information in a probability distribution with simple summaries - whether these be parameters or moments of the distributions. The extraordinary pre-eminence of the exponential distribution in hardware reliability is probably the historical reason for the obsession with the mean. At the very least it explains the confusion between mean time between failures and mean time to next failure (i.e. measured from an arbitrary time origin,

not a failure), since it is only for the exponential distribution that these are identical. There are, however, good reasons for thinking that certain complex hardware devices might follow an exponential law ([1], pp. 18-22). These reasons, unfortunately, will not justify its application to software. It remains an open question whether a failure law can be developed which reflects the structure of software, but in the absence of such a law we should beware of applying to software those hardware reliability concepts which have the exponential distribution as their basis.

## 3. Software reliability measurement

In the previous section we have seen instances where the hardware reliability experience has proved a mixed blessing for software reliability measurement. Even in situations where the hardware concepts might be appropriate, there is a great deal of confusion about these most basic definitions. This is exemplified by the following passage from an otherwise informative recent article [15]:

"Three terms are commonly used to measure reliability of a system: 'availability', 'reliability' and 'mean time to repair (or recover)'. Availability is Lefined as the probability that the system is capable of being used at time T. Reliability is defined as the probability that the time between component failures is greater than t. Thus (sic), mean time between failures is the usual measure of reliability.

"Availability is maximised by maximising the mean time between failures and minimising the mean time to repair or recovery....".

In what follows I shall try to give a personal view of some aspects of software reliability measurement. The approach may appear more critical and descriptive than prescriptive, it being easier to point to defects of existing techniques than suggest new ones. In any case it seems to me that the role of a probabilist is to describe carefully the implications of certain mathematical and probabilistic ideas and allow the customer to choose the ones which are most appropriate for his applications.

## 3.1 Bugs bared

At some risk of oversimplifying, bugs (or errors) can be defined as those 'defects' in the program which cause failures in its dynamic operation. As such, they are the things which the software engineer will try to eliminate during program testing and development. Mills [13,14] would prefer to concentrate effort on ensuring they never get into the program in the first place.

I am not convinced that the advocates of measuring reliability via bug counting have presented a strong case. It seems to me that our objective should be to measure the quality of the "behaviour" of the software, its operational reliability, rather than the quality of its "state". A "good" program is defined in terms of what it does, not what it is. The proof of the pudding.....

Of course, when the program fails in some way, it is the software engineer's job to find the cause(s) of that failure: he has to eliminate bugs. Although the bug eliminator and the reliability measurer may be embodied in the same person, the operations are quite different and only confusion ensues from identifying them. In fact I suspect that the distinction might have important economic and social consequences: would it not be a great help in getting better quality software to insist that the reliability improvement/measurement interface coincide with that of contractor/customer? I have no direct experience of US practice, but I know of cases in the UK where contractors have acted as their own judge and jury.

It can be argued that the state of a program (number of bugs) determines its performance (operational reliability), but any relationship here is likely to be very complicated and unknown. Certainly the kind of assumptions which have been made seem very naive, thus Shooman [18] says:

"...the software failure rate (crash rate) is proportional to the number of remaining errors."

I have never seen a program where this assumption could be valid. It is easy to imagine a scenario where a program with two bugs in little exercised portions of code is more reliable than a program with only one, frequently encountered, bug.

I concede, though, that it might sometimes be possible to model realistically the relationship between operational reliability and number of residual bugs. But why bother? Why introduce this extra risk of modelling error, when we can do anything we want in terms of operational reliability, directly? Thus, for example, if we want estimates of the debugging time needed to attain a specified reliability (NB specified in terms of the quantity of real interest, operational reliability), this can be achieved via a suitable model based upon operational reliability (see, for example, [II] p 113).

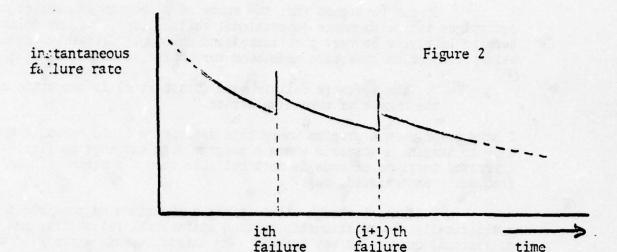
## 3.2 How to measure operational reliability

Having convinced, I hope that operational reliability is what we should be measuring, we now have to consider ways of doing this. Although the problem is a dual one of modelling and estimation, my present concern is with the former, since it is in the modelling stage that I believe we must be concerned with some quite unique properties of software.

We must consider a renewal-type process in continuous time, the successive renewals representing successive failures of the software. For simplicity assume initially that "repairs" are instantaneous (there is surprisingly little information available about repair times). It is probably worth mentioning, also, that "time" should be "execution time" (see Musa [16]): clearly there can be no failures when the program is not being used.

Such renewal processes can be characterised either in terms of their successive inter-event times, or via the numbers of events in fixed time intervals. The former method is the more appropriate one for our purposes. My contention is that the distributions of times between failures may have unusual properties, in particular their moments may not exist (viz, are infinite). If this were true then classical measures such as mean-time-to-next-failure (mttf), mean-time-between-failures (mtbf) would be infinite, and any estimates of them (although finite themselves) would be meaningless.

This assertion is quite revolutionary, and it would be pleasant to be able to say that I have evidence to support it from actual software failure data. Unfortunately this is not the case. As far as I am aware there is no good statistical test of the hypothesis that a set of data comes from a moment-less population. In any case, the nature of the problem is likely to require that such a test be based upon a large amount of data. Software failure data is still notoriously difficult to obtain in large quantities.



In fact it is quite possible that belief in a program get progressively worse if failures are sufficiently frequent, a situation which may occur in practice (see Shooman and Natarajan [20], footnote, p.164). The strength of the model lies in its ability to produce the appropriate answer automatically : reliability growth or reliability decay do not need to be input a priori. The most important advantage of Bayesian models, though, lies in their ability to reflect that attitude to programming which has Mills as its best known proponent. He states [13]: "....never finding the first error gives more confidence than finding the last error". The Bayesian, "no news is good news" property represents this exactly : periods of failure-free working cause the reliability to improve. It is interesting to compare this with the bugcounting methods of Jelinski and Moranda, Shooman et al: here reliability improvement can only take place at a failure, since it is only at such a time that an error can be removed. They are thus in direct contradiction to the Mills doctrine: increasing confidence in the program as failures occur!

# 3.4 Reliability versus utility

A surprising omission from most of the literature of software reliability is a concern for the consequences of failures in terms of economic (or other) cost. This, again, may be a result of the subject's close connections with hardware reliability, which has traditionally concentrated on modelling the failure process. The omission is rectified to some extent in the wider software engineering context: management techniques, on the whole, being cost-conscious. This literature, unfortunately, tends to be more qualitative than quantitative.

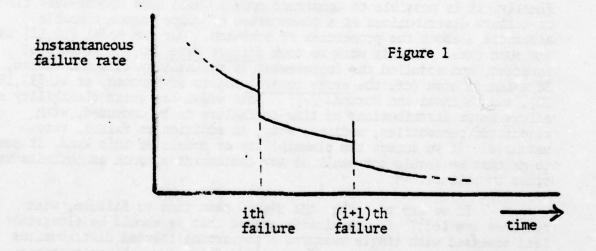
Hardware reliability tells us how to make a system as reliable as we desire by using sufficiently many components of a given unreliability. We can thus diminish the life-time cost of particular types of failure by making their frequency of occurrence sufficiently

# 3.3 Software engineers should be Bayesians

In this section I hope to convince you that we should use Bayesian interpretations and methods for software reliability. There are at least two good reasons.

In the first place, the subjective interpretation of probability, which is usually associated with the Bayesian school, seems more appropriate for software than a frequentist approach. Because of the "one-off" nature of software, there is usually no sense in which one can envisage an ensemble of repetitions of the reliability measuring operation upon which a frequentist interpretation would depend. This is, of course, an important difference between hardware components, which are essentially repeatable, and software.

The second reason is more pragmatic. The essence of Bayes Theorem is that it provides a means of continuously updating previous reliability measurements in the light of new data. Consider, as an example, the kind of calculation which is possible with our own Bayesian reliability growth model [10,12]. Figure 1 gives a portion of the plot of instantaneous failure rate which can be obtained, showing the development as time passes. Thus, prior to the



ith failure, whilst the program is working without fault, confidence in it increases and the failure rate falls. When the i th failure occurs, confidence drops and the failure rate increases, but immediately falls by a finite amount owing to the efficacy of the repair. After the (instantaneous) repair the program works again in continuous time, and during this period the failure rate again falls continuously. If our belief in the skill of the debugger is not sufficiently great to overcome our pessimism at the occurrence of a failure, then a plot such as Fig. 2 will result.

Support for this idea, then, must be analytical rather than directly evidential. In the first place it could be asked, slightly frivolously, what evidence there is for the existence of morants. Those people who wish to estimate mttf should be asked to furnish evidence that the quantity they are estimating does indeed exist. One can always obtain a finite average of some data, after all, but it may not estimate a population mean. More seriously, there is the unique property of software that it suffers no natural degradation: once perfect, it will never fail. If we concede, therefore, that there is some chance (however small) that the program is perfect, then the mean time to failure must be infinite (see [6]). In fact this case is so extreme that not even fractional moments would exist. But is it such an extreme assumption? When we come to look at the problem of incorporating structural information into our reliability modelling we shall consider modular programming - it does not seem unreasonable to believe that a small enough module might be perfect. In fact Mills goes much further, a serting that top-down structuring is likely to produce perfect programs, even when these are large [14]:

"The new reality is that you can learn to consistently write programs which are correct ab initio, and prove to be error free in their debugging and subsequent use."

Finally, it is possible to construct models which have moment-less time-to-failure distributions as a consequence of quite unexceptionable assumptions about the properties of software. Our own model [10,12] is one such case. In this work we took failure rate as the measure of interest, and modelled the improvement of reliability in a subjective, Bayesian fashion (cf. the error removal models of Shooman, et al [17,18, 20], and Jelinski and Moranda [5]). This model has great flexibility and allows exact distributions of time-to-failure to be computed, with associated percentiles, medians, etc., in addition to failure rate measures. If we accept the plausibility of models of this kind, it seems to me that we should not bauik at any consequences, such as infinite mean times to failure.

If we are to escher the use of mean time to failure, what measures are left? I have already argued that we should be altogether less obsessed with single measures: preferring instead distributions from which we can calculate, if required, many appropriate measures. Thus from a time to failure distribution we could quote the probability that the time to next failure exceeded any required value. This enables us to tailor our measures to the particular situation in hand. If a single measure of quality is really required, failure rate (hazard rate) is more plausible than mean time to failure, since it will exist under much wider conditions. Indeed, for the decreasing failure rate [1] situations which we hope to encounter in software, instantaneous failure rate has a more natural intuitive interpretation than (instantaneous) mean time to failure.

Notice that these considerations about the possible non-existence of the mean time to failure may cause difficulties with the definition of availability, as mentioned earlier. Even if the mean does not exist, it may be that the fraction of time available converges in probability to some constant, but this cannot be assumed. In fact detailed knowledge of the distributions of time to failure and time to repair will be needed. Scant attention seem to have been paid to repair time distributions in the literature.

COMPUTER SCIENCES CORP ARLINGTON VA

SOFTWARE PHENOMENOLOGY - WORKING PAPERS OF THE SOFTWARE LIFE CY-ETC(U)

AUG 77 B ELKINS, L HUNT

DAHC26-76-D-1006 AD-A053 014 UNCLASSIFIED NL 6 OF **8**053014 題

small. Since we do not yet really have such techniques available for software (and we may never get them), it seems to me particularly important that we at least estimate life-time costs of our systems. In fact, I wonder how many projects we would embark upon if we knew how much they would cost over their life-time, rather than merely to create.....

It does not matter much whether we decide to adopt utility (profit) or cost as our criterion : a choice will usually be dictated by personality and circumstances. Thus Buzen et al [2] compared the utilities of virtual machine and conventional operating systems : in fact they assume constant, linear utilities. In my own work I have, pessimistically, used costs [9]: the system can fail in different modes, each particular failure resulting in a (random variable) cost with a distribution typical of the mode. Interest then centres upon the total cost of running the system for time T. Clearly this is a random variable, and, in order to be true to my evangelism of earlier sections, it would be most informative to obtain its distribution, in particular as  $\Gamma + \infty$ . This can, in fact, be done. In practice, of course, the system will "earn" utility during its working periods and "lose" during failed periods (cost of repair, loss of revenue, etc.). This refinement does not seem difficult to achieve in both the above models.

It is often argued, against this kind of approach, that we hardly ever have much information about costs. This seems unnecessarily defeatist: after all, merely counting failures is equivalent to giving a unit cost to each. Surely we always know better than that?

## 4. Structural models

Most of the earlier part of this paper was concerned with measuring the quality of a single, "black-box" program: the equivalent of a device or component in the hardware terminology. Fortunately, we normally have a large amount of information available about the structure of the program, and it would clearly be sensible to use this in our reliability modelling. This does not, however, seem to be an easy task. Most attempts so far have looked at fairly specialised structure, and I shall briefly review two of these in what follows, but the real goal is the modelling of more general systems.

One of the most interesting attempts to go beyond the black-box approach is that of Buzen, et al , [2] who studied virtual machine (VM) techniques. They define a  $\overline{\text{VM}}$  as :

".... a hardware-software duplicate of a real existing computer system in which a statistically dominant subset of the virtual processor's instructions execute directly on the host processor in native mode.

"Thus a VM provides for the efficient operation of one or more copies of a complete computer system which is similar to the host (or real) system."

The program which mediates between the virtual machine and the actual resources of the system (host machine) is called the virtual machine monitor (VMM). This provides an extremely high degree of isolation for each VM running under its control, so that a failure in one VM will not affect the operation of other VM's. Although a VMM failure will interfere with the operation of all VMs, there is good reason to believe that the VMM can be made very reliable. The example used in [2] is a system which supports N levels of multiprogramming. The usual method is to construct a multiprogrammed operating system (OS-N) which runs on a bare machine and supports N independent user programs. The virtual machine approach is to run N copies of a monoprogramed operating system (OS-1) under control of a VMM. By assuming all recovery times and times between failures to be exponentially distributed, the authors are able to prove that the expected utility of VAM/OS-1 is greater than or equal to the expected utility of OS-N, with quite weak assumptions on the means of the exponential distributions. In fact I have been able to generalise their result to the case where failure and recovery times are arbitrarily distributed, with means (unpublished work). This generalisation makes their result very powerful and is a compelling argument for the VM approach in such situation.

My own work on structural models [7,8,9] has been Markovian in nature. The assumption here is that the program can be decomposed into R subprogram (modules) and execution proceeds by switching among these according to a Markov (or semi-Markov) process. Each module is assumed to fail in a Poisson process (i.e. exponential inter-failure times), with different modules having different failure rates. A record of the overall failure process is thus a sequence of failures attributable to the different modules, and its probabilistic description is exceptionally complex. Fortunately, simple limiting results can be obtained when, as seems plausible, the failurates are very much slower than the switching rates between modules. It call

be shown, for example, that the overall failure process (i.e. merely counting failures and not attributing them to a source module) is Poisson, with an easily calculated rate. The model can be extended to allow each failure to have a (random variable) cost, particular to its module, and the limiting distribution of the total cost in running the program for a time T can be obtained. The parameters for these limiting distributions are quite easy to estimate. For example, by flagging the program it should be possible to obtain good estimates of the transition probabilities of the embedded Markov chain very quickly. Thus the model could genuinely be used as a design tool: enabling the system reliability to be calculated from the module reliabilities (obtained from testing these individually) together with the knowledge of how they interact dynamically. Of course, models like this are intended to be quite general and flexible, so a measure of their quality will always be the appropriateness of their underlying assumptions. The two basic assumptions here are that the switching be (semi-) Markov, and each failure process Poisson. The second is quite plausible and, at least for the limiting theory, could possibly be relaxed. It is in the <u>limited memory</u> of the Markov assumption that problems are likely to arise. It may be possible to test this in some cases, although this needs the system to be running, which may defeat the purpose of the exercise. On the other hand, it seems likely that we could sometimes make this assumption a priori, for example in realtime situations where the program is reacting to a random data input stream. Even if the process is not exactly Markovian, all is not necessarily lost: it may be possible to make it Markovian, albeit at the expense of a greatly enlarged state space.

Still, the chief criticism of this model is that its assumptions come from a desire for mathematical tractability rather than software plausibility. Unfortunately, the kind of structure which is natural to software tends to be of a static kind: the program as-it-is. Our requirement for operational reliability demands that we define structure in a dynamic way: the program as-it-performs. The ideas of top-down structured programming might be able to bridge this gap, but the way forward is not clear at present.

However, in spite of all this, it is beginning to be obvious that techniques such as modularity, top-down structuring, etc., are gaining in popularity for good practical reasons : they appear to deliver the goods. If they do become more widely utilised, the following intriguing question becomes of interest : do these methodologies inevitably result in the final program having a particular failure law? The case of top-down structuring is particularly interesting in view of the simplicity of the structure - three basic program figures [13]. Alternatively, in modular terminology, where we imagine a program to be comprised of subprograms which in turn are comprised of subsubprograms, etc., what is the relationship between the reliability of a module and the reliabilities of the submodules it comprises? It seems likely that modules and submodules have the same failure law of necessity (with, presumably, different parameter values), since the methodology used in their creation would be similar. What would such a failure law be? An answer in a special case is given in my Markov model, where it is proved that if the subprograms follow a Poisson law, then the program itself will (in the limit). It would be of great interest to see whether there are other cases where the form of the failure law is dictated by program structure.

## 5. Summary and recommendations

- (i) Do not apply hardware techniques to software without thinking carefully. Software differs from hardware in important respects: we ignore these at our peril. In particular
- (ii) do not use mttf, mtbf for software, unless certain that they exist. Even then, remember that
- (iii) distributions are always more informative than moments or parameters, so try to avoid commitment to a single measure of reliability.
- (iv) Software reliability means operational reliability. Who cares how many bugs are in a program? We are concerned with their effect upon its operation.
- (v) Use a Bayesian approach and do not be afraid to be subjective. On the contrary, all our statements will ultimately be about our beliefs in the quality of programs.
- (vi) Do not stop at a reliability analysis: try to model the lifetime utility (or cost) of programs.
  - (vii) Now is the time to devote effort to structural models.
- (viii) Structure should be of a kind appropriate to software, e.g. the top-down structuring of Mills [13,14].

Finally, a plea for more validation of models. We can argue the relative merits of our particular pet schemes for ever, but they must ultimately be tested in the real world. Would anyone like to support a competition between rival models, providing suitable data to be analysed by the different methods? Would anyone like to enter such a competition ....?

## References

- ! 1] BARLOW, R.E. and PROSCHAN, F., Mathematical Theory of Reliability. New York: Wiley, 1965.
- [ 2] BUZEN, J.P., CHEN, P.P. and GOLDBERG, R.P., 'Virtual machine techniques for improving system reliability', in Record, 1973 IEEE Symposium on Computer Software Reliability, New York, N.Y., Apr. 30-May 2, 1973, pp. 12-17.
- [ 3] CHUNG, K.L., "A Course in Probability Theory". New York: Harcourt, Brace and World, 1968.
- [ 4] COX, D.R., "Renewal Theory". London: Methuen, 1962.
- [5] JELINSKI, Z. and MORANDA, P.B., "Software reliability research", in Statistical Computer Performance Evaluation, Ed.: W. Freiberger. New York: Academic, 1972, pp. 465-484.
- [ 6] LITTLEWOOD, B., 'MTBF is meaningless in software reliability', (letter) IEEE Trans. on Reliability, April 1975, p82.
- [7] LITTLEWOOD, B., "A reliability model for Markov structured software", in Proceedings of the 1975 International Conference on Reliable Software, Los Angeles, Cal., April 21-23, 1975, pp. 204-207.
- [8] LITTLEWOOD, B., "A reliability model for systems with Markov structure", Applied Statistics (J. Royal Statist. Soc., Series C) Vol.24, No.2, 1975, pp. 172-177.
- [ 9] LITTLEWOOD, B., "A semi Markov model for software reliability with failure costs", in <u>Proceedings of the Symposium on Computer Software Engineering</u>, New York, N.Y., April 20-22, 1976, pp. 281-300.
- [10] LITTLEWCOD, B. and VERRALL, J.L., "A Bayesian reliability growth model for computer software", Applied Statistics (J. Royal Statist. Soc., Series C) Vol. 22, No.3, 1973, pp. 332-346.
- [11] LITTLEWOOD, B. and VERRALL, J.L., "A Bayesian reliability model with a stochastically monotone failure rate", IEEE Trans.on Reliability, Vol. R-23, No.2, June 1974, pp. 108-114.
- [12] LITTLEWOOD, B. and VERRALL, J.L., "A Bayesian reliability growth model for computer software", same source as [2], pp. 70-76.
- [13] MILLS, H.D., 'On the development of large programs', same source as [2], pp.155-159.
- [14] MILLS, H.D., "How to write correct programs and know it", same source as [7], pp. 363-370.
- [15] MDRGAN, D.E. and TAYLOR, D.J., "A survey of methods of achieving reliable software", Computer, Vol.10, No.2, February 1977, pp. 44-53.

- [16] MUSA, J.D., "A theory of software reliability and its application", IEEE Trans. on Software Engineering, Vol.SE-1, No.3, September 1975, pp. 312-327.
- [17] SHOOMAN, M., "Probabilitic models for software reliability and prediction", same source as [5], pp. 485-502.
- [18] SHOOMAN, M., "Operational testing and software reliability estimation during program development", same source as [2], pp. 51-57.
- [19] SHOOMAN, M., "Structural models for software reliability prediction", in Proc. 2nd International Conference on Software Engineering, San Francisco, Oct. 1976, pp. 268-280.
- [20] SHOOMAN, M. and NATARAJAN, S., 'Effect of manpower deployment and bug generation on software error models', same source as .9., pp. 155-170.

A SUBJECTIVE EVALUATION OF SELECTED PROGRAM DEVELOPMENT TOOLS

#### ABSTRACT

A program development project can be viewed as a complex system of people, of hardware and software, of tools. Any one element should be viewed in its system context. With this in mind a selection of analysis, design, development, and test tools is discussed. Some results of completed analyses are presented; however, the interpretation of the results is subjective based on the author's experience and opinions. The material in this paper is part of a book chapter. The chapter references are included in toto. Paragraph numbering follows the chapter sequence.

J. D. Aron
Tour Franklin, Cedex 11
92081 Paris, La Defense, France

### Copyright 1977

This material will appear in a forthcoming book to be published by Addison Wesley Publishing Company. No copies of this material may be made without the author's permission. A tool is designed to be used for a specific purpose and it must be handled according to instructions. Improperly used it can cause more damage than it is worth. Carpenters learn this early when they are taught how to use a tool such as an adze for squaring off logs. The adze is built something like an axe with a horizontal cutting edge. The carpenter straddles the log and wields the adze in an arc, following through between his legs. Properly used, the adze rapidly produces a neat plane surface on a large log or plank. Improperly used the adze is dangerous. A misstroke can cut too deep and spoil the surface beyond recovery. More important, a loose grip, a glancing blow, or bad aim can turn the cutting edge into the carpenter's leg with bloody consequences. The carpenter has to be convinced that the benefit of using an adze exceeds the cost of buying and caring for one and, in addition, exceeds the cost of a mistake. Not surprisingly, the adze is found in relatively few carpentry kits.

Tools used in programming projects are similar to tools in a carpenter's kit. Each has an initial procurement cost. Each has a continuing maintenance and update cost. Each requires proper training before use. The project must be adjusted somewhat to fit the tool. And, in spite of the precautions taken by the toolmaker and the user to avoid misuse, the tool can cause irreparable damage. A tradeoff analysis comparing the costs and potential risks to the potential benefits is a prerequisite to a commitment to a specific tool.

There are three dominant reasons why managers authorize the use of tools:

- o increase productivity
- o improve quality
- o improve resource and schedule control.

Really, all three are only different aspects of a single reason for justifying the expense of a tool; namely, that the value exceeds the cost.

Tools generate value by increasing the useful output obtained from a given unit of work. For a program systems costing \$30,000,000 containing one million lines of code the cost of production would be \$30 per thousand lines of code or \$30/KLOC. Programmer productivity for this project may

The scale factor KLOC has been recommended for measuring many aspects of program development because it is based on a measurable characteristic of actual programs. Although "lines of code" is defined in various ways (to include or exclude comments, to include or exclude instructions resulting from macro expansion, etc.), the number of lines in a specific program can be determined simply by counting. The cost elements of the project that produced that program can be stated as \$/KLOC, \$\tilde{\text{NLOC}}\$ son-months/KLOC, defects/KLOC, etc. facilitating comparisons across products. A key supporter of the KLOC unit of measure, Capers Jones of IBM, has shown it to be the most useful common denominator of the many measures used in IBM. [9]

be 2500 instructions per programmer-year or 0.4 programmer years/KLOC. (At \$30,000/person-year, 1000 person-years expended, 400 programmer-years, 600 support-and manager-year.) A similar project using a tool that improves the productivity of programming and testing 25% should spend proportionately less. Allocating the 25% saving to programming and the test portion of support (200 tester-years) and leaving everything else the same, the project would use 850 person-years costing only \$25,500,000 plus the cost of the aid. In most organizations, an aid costing as much as \$4,500,000 would be attractive if it could ensure the indicated personnel savings.

An improvement of 25% in programming and testing is a very large improvement. Most aids do not promise that much; in fact, most aids do not promise any improvement. The value of an aid is almost always based on consensus opinion rather than on quantitative factors. Each aid is promoted by its originator on the basis of common sense and demonstration. Thus, a technique such as structured programming is presented as "a logical way to write programs". A program is presented in two versions, one structured, the other not. If it is not obvious to the observer that the structured version is better, the demonstrator will ask a few questions about the program's function. The observer will find it easier to answer the questions by reading the structured version. In time the observer will convince himself that structured programming has merit, arriving at this conclusion at about the same time as the bulk of his colleagues. Most techniques gain acceptance in a similar way. Initially, someone originates the technique to solve a practical problem. It works; so the originator or an associate uses it, becomes devoted to it, and proceeds to promote it outside his immediate area. For a while, there is little response other than admiration for a resourceful problem solver. Gradually, a few more people independently pick up the technique until a ground swell of enthusiasm sweeps the technique into wide use. At this time, each new user adopts the technique because "everyone is using it; it must be good". And so it is. The technique would have been abandoned early if it were inadequate. Interestingly, it is usually after the widespread acceptance of the technique that people start to measure its effectiveness. The data obtained can be used to suggest areas for improvement but are too late to affect management decisions regarding the technique's value.

The pros and cons of each tool an individual might elect to use are reasonably self-evident. The tools used by individuals on medium and large projects (including many already discussed) interact in more complex ways and may be harder to evaluate. For instance, an on-line programming system which allows a programmer to type in a source program, edit it interactively, and compile and run it without leaving the office may cut many days off the schedule for producing the unit of programming. Yet, with a hundred programmers on a single project, the benefits of the interactive terminal may disappear. For one thing, it may not be economically feasible to give everyone a terminal. This may cause queues to build up at each terminal and may force some people to leave the

office to find an available terminal. This takes away much of the convenience and immediacy of on-line operation. A project that depends on interactive methods with no backup technique may be out of business when the support computers are not available.

Another factor affecting the large groups may be the inability of one computer to handle the whole workload. As will be seen later, the integration of a large program system using classical methods chews up many hours of computer time. The interactive terminal user generates the heavy integration and test workload by submitting completed program units for integration. He then must compete with the integration workload in order to get time for editing and debugging interactively the program unit he is currently working on. When the workload fills the computer, the terminal response time gets worse and worse until it is no longer efficient. When the workload overflows the computer, a new problem develops: there is no longer a convenient way to refer to all parts of the system. A new communication and control procedure is needed to bridge the gap from one computer to the other. This is a quite difficult and expensive step; yet, without it, the basic interactive terminal loses some of its. capability. It's like a two-man saw. It works fine on small trees where one-man can get on each end of the saw. But very large trees can't be spanned by the saw so it may be no use at all.

On large projects, the set of tools that was appropriate for small jobs must be augmented by additional tools designed just for large projects. Most such tools emphasize standard ways of doing things and standard ways of describing things so everyone on the project can understand what is going on and so deviations from the plan are easy to spot.

There are three areas in which special tools can be important on a large project:

- o analysis and design
- o development and test
- o status control

Various large project tools are covered in later chapters in sufficient detail to show what they are used for, how they affect the project, and what tradeoffs should be made when evaluating them. In most cases, the examples are adequately described elsewhere; therefore, rather than explain each tool in detail, references to the literature are provided. The general state-of-the-art of software engineering in 1976 is discussed by Boehm [10].

2.3.1 Analysis and Design Tools

The steps in a project (Table 2.1) proceed from a general problem statement to a more specific requirements document showing exactly what the system must do. In order to help relate the various relevant aspects of the

problem to one another, analysts can draw on techniques for thinking about complex problems. One class of techniques is structural modeling [11]. It uses various methods of showing the relationships among problem elements and weighting the relationships so as to elucidate the structure inherent in the problem. Given the structure, system dynamics modeling [12] can be useful in pointing out behavioral aspects of the problem. The analysts who develop the detailed requirements select what they believe is the minimum set of necessary, feasible items. System architects respond to the requirements by specifying what will be built - often a subset of the requirements statement - and how it will look to the user. The purpose of this step is to specify the external interfaces of the system and to allocate system functions to either side of the interfaces. Architecture is key in systems which contain or communicate with existing subsystems which constrain the design of the new system in order to preserve compatibility. From this point, the internal design of how the new system will behave can proceed step-by-step. design is detailed enough to identify program units, the units are assigned to individuals who build them and submit the results for integration, test, and release to operations, followed, in most cases, by a period of maintenance and improvement. The large number of people involved in these activities makes it very difficult to have direct communication between all of the implementers and the key analysts and designers. The analysts/designers, knowing that they are writing specs for people they may never meet, need a way of writing requirements/specifications that is unambiguous and easy to understand. The method should be particularly good for describing interfaces. It should be easy to maintain and modify and should be in a form that supports simulation. The components of such a tool are an analysis/design language, an automated specification program, a simulation capability, and a methodology for ensuring complete, consistent results. Everyone must learn the language (which may be a programming language or simply a uniform way of writing specs). The other components are used mainly by the analysts/designers. The automated spec is a means of monitoring programs written by the implementers to see that they conform to the spec. The output of this program can go back to the implementers to help them see where they went wrong. The simulator exercises a model of the system to check that, if all the specs are followed, the system will work. It is the designers' proof to the implementers that the spec is valid and should be obeyed. Now, even though no single designer or implementer understands the whole system, there is a single baseline spec that everyone can refer to in order to resolve questions.

2.3.1.1 Analysis and Design Languages

Languages for system analysis must work in the managerial as well as the technical areas of system development. They must cope with poorly defined user requests, unsolved technical problems, and arbitrary resource constraints. The indefinite nature of these aspects of analysis tends to result in more text segments - explanations of assumptions or justification of decisions - they are found in finished programs. Still, to improve clarity, traceability, testability, and to maintain positive control of the contents of the requirements at all times, system analysis languages have been developed. One example, based on the Problem Statement Language/Problem Statement Analyzer (PSL/PSA) originated

by the University of Michigan ISDOS project [13], is contained in REVS [14]. REVS is a requirements engineering and validation system developed by TRW for use in large real-time systems. The language component of REVS is RSL. RSL deals with system entities, messages, activities, and networks (R-nets) formed from the possible paths messages may follow from an input interface to an output interface. The R-nets are constructed according to a small number of rules for where to start and how to proceed. The result is a requirements statement document which can be analyzed for completeness by the REVS support facilities. The specific TRW system has added capability for building and executing simulations from the requirements and evaluating the feasibility of meeting the system performance requirements. All of this is embedded in a REVS library facility which has a flexible inquiry capability permitting managers to investigate project status from a variety of viewpoints. It is significant that REVS includes "engineering" in its title. Experience with the hardware portions of large systems led engineers to adopt formal procedures for defining the project. REVS carries engineering discipline over to software activities.

A second analysis approach is SADT, the structured analysis and design technique of SofTech, Inc. Devised by Ross [15], SADT offers a graphic language, as it were, to show the functions to be performed by a system. Great emphasis is placed on the identification scheme used to label activity or data boxes and the arrows which connect the boxes, showing inputs, outputs, controls, and proposed mechanisms for implementation. As a result, SADT permits multiple hierarchical views of the system to be correlated with one another. Each viewpoint supports one important aspect of requirements analysis. It is often easier to develop the requirements in one class if the requirements of another class are separated out. Thus, software requirements to satisfy all members of a user's group may provide one viewpoint and a huge list of functions to be performed. Different requirements as presented by a developer reflect concern for compatibility, economy, useability, and other factors which make up a much smaller list of functions. SADT permits both analyses to proceed and then provides the means for bringing them together in order to decide on the list of functions to be adopted which may be a compromise between the two viewpoints. The reference system that supports multiple views also provides the data needed to check the requirements document for completeness and consistency. SADT differs from REVS in that the analysis and design stages of a project are both supported by SADT. In fact, SADT methodology encourages iteration between requirements analysis and design at appropriate points. One such point is when analysis stops at an obstacle which could be removed by a design decision. For example, detailed requirements for the content and visual quality of a display may depend on knowing what physical device is to be provided by the implementer.

Generally, all analysis and design techniques including SADT postpone such decisions as long as possible. SADT, however, is very flexible in this respect whereas some design techniques assume that analysis is complete before design starts. In the latter group are the design methodologies strongly influenced by the structure of commercial information systems departments. Analysis is often done by a different part of the organization from design in commercial firms. The analyst skills stress subject area experience (financial, personnel) and design skills stress data processing experience. The departmental split permits economic justification for new work to be done by the analysts to avoid make-work tasks originating in the programming department. Within the programming department

there is a need for tools which raise the performance of all the staff up to an acceptable average. For this reason, design approaches such as Jackson's [16], Langefors'[17], and Warnier's [18] give rules for converting from the design language to actual programs. More will be said about these approaches in Chapter 6.

Languages exclusively for program design have not achieved wide acceptance. Unlike procedural languages such as PL/I or COBOL, design languages do not have an agreed-upon core vocabulary. As a result, there is no one design language that everyone recognizes as the best. APL comes close because many people know it. Nevertheless, APL is used more to evaluate designs and to create models for analysis than it is to document designs. The power of set operations available in APL is the basis of SETL, advanced by Schwartz [19] as a program development language. Less comprehensive proposals such as the Module Interconnection Language (MIL) suggested by DeRemer and Kron [20] tackle less of the development process, concentrating in this case on formalizing interface designs. It may be that no standard design language will ever exist because it is not needed. The evidence for this comes from the increase in informal program design languages which mix natural language (e.g., English) descriptions with statements in a high-order structured programming language (HOL) such as PL/I, ALGOL, structured COBOL, or structured FORTRAN. The informal approach builds on the work of Dijkstra [21, 22], Wirth [23], Mills [24] and others whose emphasis on top-down development makes good use of mixed format. The most successful informal languages or pseudo-codes establish a small number of key words/ commands which are, in effect, the names of the basic control structures of structured programming [25]. Design is more concerned with the flow of control and data among program units than with the internal behavior of the units; therefore, it is natural and effective to use a design language that highlights control structures. Flow can be verified by deleting the natural language and compiling the remaining statements. It is a straightforward step to proceed from the pseudo-code design to a formally coded program.

### 2.3.1.2 Automatic Specification

This choice of an informal design language makes it more difficult to build an automated specification tool. In order to verify that a program agrees with its spec, the tool must be able to recognize corresponding names, dimensions, displacements, and the like. Informal design languages do not have built-in enforcement procedures to require all programmers to use the same conventions. Obviously, rigorous rules take away much of the informality. Sooner or later, however, all references to the same object in a system must be reduced to a common base. It is a function of design control to see that this happens. A compromise approach that seems feasible is to rely on a self-documenting, compilable design as the input to programmers and to automated spec tools. Informality is preserved up to a point. The designer can use any technique for draft specifications. The final specification must be free of natural language and contain only legitimate statements in a programming language. The spec is selfdocumenting if appropriate comment statements have been included (in the working code or, better, in a prologue). It is compilable because it uses HOL statements exclusively. At this stage, all system references must agree with the standard references in the glossary of system names and synonyms. As a design, however, the spec lacks the detail necessary to make the program units function.

Now an automated spec tool can compare a completed program unit to the spec and determine that, for each control statement, identifier, or variable in the spec, there is a corresponding, correctly named entry in the program. It can also flag program references that have no counterpart in the spec. Deviations such as this are brought to the attention of both the programmer and the designer (or design control group) to ensure that the deviation is explained or corrected [26].

As more is learned about proving the correctness of programs, it is likely that both designs and coded program units will be checked for correctness. To do this, assertions regarding the intended behavior of a program must be included in the spec. A proof mechanism built into the automated spec tool would test the assertion by carrying out the logical steps described by the spec or the completed program. The results would be particularly useful in large programs with many internal branch points. It is easy for the designer to get confused in such circumstances. Unfortunately, the state-of-the-art of correctness proofs in the 1970s does not span large enough program systems to solve the designer's problem [27]. Future promise of proof tools nevertheless justifies the adoption today of the discipline of making appropriate assertions as an aid to program verification. Once made, the assertions will be a useful aid to inspectors who examine the design and code at appropriate points in the project. The assertions will clarify the programmer's intent better, perhaps, than the program text.

#### 2.3.1.3 HIPO

A simple procedure for preparing design specs so they are readable, wellorganized, systematic, and useful to programmers in HIPO [28]. HIPO is a documentation technique for displaying Hierarchy, Input, Processing, and Output. It parallels top-down design by proceeding from overview to detail (Fig. 2.8). Two types of charts are used. One is a tree structure representing the program system structure (Fig. 2.9a). It is the index to the process diagrams (Fig. 2.9b) which show inputs, process, and outputs and below if necessary additional information (such as assertions, management guidance as to schedules, resources, etc., or technical guidance such as standards or macros to be used). The detail diagrams can be filled in with English or pseudo-code or POL statements. The purpose of HIPO is to provide a graphic view of a complex system without getting lost in the detail. HIPO does not give guidance in how to design. It simply documents the design. In this respect HIPO is less sophisticated than the design techniques already mentioned. It is also simpler and cheaper. Being easy to use in any environment, HIPO can help overcome the communication problems that often hurt projects.

HIPO is quite similar to an approach taken by the System Development Corporation when they prepared a planning guide for the use of the U. S. Naval Command Systems Support Activity [29]. Prepared in 1965, the guide gave Navy project leaders instructions for managing a complete programming life cycle. In most respects, it is valid today. It is mentioned here because the main technique for showing the steps in the life cycle is remarkably like HIPO. The life cycle is treated as a sequence so no tree structure diagrams appear; however, each detail diagram looks like a HIPO chart with the addition of cost and environment data (Fig. 2.10). It is clear that the HIPO technique is useful and flexible and, indeed, meets a need for more systematic design documentation [30].

### 2.3.1.4 Modelling & Simulation

A design language and some means of design verification give a degree of control over the functions of the system being built. These tools do not deal with system performance. Modelling and simulation are required to predict performance. Modelling consists of building an analytical or a procedural model of the system design. Simulation consists of feeding realistic inputs to the model and observing the model's behavior.

In the early stages of design, analytical models can describe the intended behavior of various portions of the system. An analytical model is a set of mathematical equations and logical expressions which represents system structure and data flow. The algorithms used to develop the equations are often patterned after existing systems that, hopefully, resemble the planned system. Thus, the paging behavior of a current operating system may be adopted to describe how memory/disk traffic will look.

Analytical models represent the design in equations which, if correct and solvable, produce repeatable results rapidly. In general, the functions of a single program unit can be modelled analytically but a program system cannot. There are too many branches in the system resulting in random (or stochastic) behavior that can only be guessed at. Also, the programs consist of discrete events which the analytical model approximates by continuous equations. As a result, analytical models alone are unsatisfactory predictors. They are best used as components of a procedural model or as quick tests to obtain ballpark estimates of performance. The latter are very important since it is sufficient to know that a design is in the right ballpark when you must decide to accept it or reject it. The greatest strength of analytical models is in predicting the effect of a design change on an existing, well-documented, communications-oriented system. In this situation, response time is a key criterion. Good data about input arrival times and subsystem service times often exists in running systems so that reasonable assessment of a design change is possible. The same methods are less useful in a new system because of the unavailability of good arrival and service time distributions. One usually assumes some standard arrival and service time distributions and settles for ballpark answers. The margin of error is high but the information obtained will still help guide the designer to better solutions.

An accepted design must be controlled to tighter tolerances. Here is where the procedural model comes in. It is constructed exactly like the system design. For each element in the design there is a corresponding element in the model and for each control sequence in the design there is a corresponding control sequence in the model. In fact, if the design is written in pseudo-language or POL, it is already the skeleton of the model. To complete the model, all you do is enter performance factors for each element or control sequence. Early in the design stage, it is necessary to guess at the performance factors. Educated guesses as to the path length, real memory size, and virtual working set size of each section of the program are best made by using similar existing programs for reference. As the project advances, the early estimates can be improved by the programmers and, eventually, exact measurements can be made of completed program units. The model gets better as it matures. During early design, results provide only

ballpark figures, perhaps 50 - 100% from the true values. By the time the system is ready for delivery, simulation can predict consistently within 5-10% of actual performance. In complex systems, simulation results may sometimes surprise the designers because they exhibit behavior due to subtle system interactions the designers never thought about. Guest, Lai, and Loyear, in their report on the extensive simulation of the intelligent controller subsystem in a banking system [31], show that such surprises are usually helpful.

The main drawback of modelling and simulation is its high cost. Even when the original design serves as the model there is a large cost associated with obtaining performance factors, updating the model, and running some number of simulations. There are only a few people doing system design so that the added cost of system analysis during the design stage appears to be doubling project cost. In fact, like most of the best tools, simulation requires an early expenditure to avoid a much larger expenditure later. Thus, \$20 - 50000 worth of systems analysis which selects a design alternative that does the customer's job can avoid \$1,000,000 worth of rework on a system that fails to pass its acceptance test. Chapter 6 gives some examples of the types of questions simulation can answer that justify its cost in complex systems.

2.3.1.5 Benchmarks and Prototypes

Managers who are frightened by high initial costs sometimes try to get performance data by other means. The most popular method is the benchmark [32]. This is not a recommended design tool. Neither is it inexpensive. A benchmark consists of a defined set of jobs with input and database given and output either known or easily checked. A benchmark test is a timing run in which the standard job stream is executed against the clock. Obviously, to be valid the test must execute actual programs. But there are no programs available during the design stage; therefore, a benchmark at this stage must be built of programs from some earlier system. The only condition under which the benchmark will predict the performance of the new system is that the new system will be identical to the old one. Yet, if you know that to be the case, you do not have to run the benchmark at all. A prototype is a small version of the planned system [33]. It may be a subset of the planned system which merely omits functions. In this form, it is often used to check a particular design decision at the module level. The prototype may also be completely different from the planned system - a "quick and dirty" version - which is going to be discarded after it is studied. Where the new system is different from the old one, two arguments are advanced for using benchmarks. Both are faulty:

- o it is possible to extrapolate from old data. The only case this applies is when the new system has the same structure in detail as the old one but has program units of different size. New systems of this type are unrealistic investments. There probably are none.
- o it is possible to extrapolate from a prototype of the new system; namely, actual code of a limited portion of the new system. This approach works to a limited degree when the prototype represents the bulk of the execution time (say, 80-90% of the path length) of the new system. Few complex systems possess such a tight kernel.

The accuracy of simulators early in the design stage is not actually known because there is never a fully executable version of the system represented at that time in the model.

2.3.1.6 Extrapolation of Software Data

The real reason these arguments fail is that extrapolation in the software business is very unreliable. When you use program A as a reference to predict something about program B, you must rely on A and B being largely alike. But, as noted above, no one wants to spend a lot of money to get a B that is just like the A he already has. B must have added value due to more functions or better cost/performance. Usually, B is either larger than A or different from A. Fig. 2.11 illustrates that, while you could predict B from A if they were alike, you cannot predict B from A when they differ, and you cannot predict B accurately when it is larger than A. The last case is the most common and the most likely to cause expensive errors. The picture shows B twice as large as A but, following the square law of complex systems, B is four times as complicated as A. In a sense, when you use A as a reference, you are ignoring 75% of the problem. This picture applies whether you are predicting performance, estimating workload, or setting schedules. Overreliance on an existing reference system will lead to large understatements of results.

A couple of examples show what has happened in the past:

- o A is similar to B one case where it is worthwhile to copy A is in the compiler business. An experienced crew using the same compiler design to produce compilers for different machines can generally achieve performance proportional to the speed and storage of the target machines.
- o A is different from B at one stage of development, the real time operating system (RTOS) for Apollo was about the same size as OS/360. RTOS used a special algorithm for routing urgent traffic whereas OS/360 made its full function available to all transactions. RTOS appeared to execute faster than OS/360 but, in fact, the two systems were not directly comparable. Each had its own user environment.
- A looks like B but is actually smaller a project to convert an existing program from a small machine to a larger, newer machine expected to get a performance improvement of 4:1 based on the ratio of machine speeds. Additional functions were added and the new program was written by inexperienced coders. It ended up 5-6 times larger than the old program. The actual performance on the larger machine was 1/10 that of the small machine.

Benchmarks are poor design tools because they are poor representations of the system to be built. When used as aids to computer selection, this major disadvantage gets cancelled out. When several bidders are given the same problem, they all start from scratch. The best test performance, while a poor indicator of ultimate system performance, may be an acceptable indicator of the best proposal. In this case, the benchmark is not expected to be a precise model of the new system. As long as it is in the same class (A looks like B), it can be used to distinguish among bidders. Thus, it is possible for government agencies and computer companies to build a library of standard benchmarks to be used for comparing one generation of computers to a predecessor or to compare one line of equipment to a competitive line [34].

As a design and development tool, modelling and simulation are substantially more useful than any level of benchmarking or prototyping as long as the model is faithful to the design and is kept current. This is feasible when the model is the hyproduct of design. The model can be kept current as a byproduct of design control. In these circumstances, one use of the model (say, 20 runs to evaluate alternate design proposals) can be expected to take 10-15 analyst-weeks and 30-60 hours of CPU use (not counting connect time when interactive terminals are used during simulation). One benchmark takes about the same number of analyst weeks and requires considerably more computer time - on the order of 100-200 hours of the main CPU and almost as much for a driver (to supply an input stream from a second CPU). The development cost of a model is spread across its life which can extend for the life of the product it represents. In fact, the most profitable use of models is often to tune or upgrade installed systems. The development cost of a benchmark is a one-time charge incurred for each benchmark test unless a library standard is used.

- 2.3.2 Development and Test Tools

  Two major types of development and test tools are important:
  - o tools that increase individual productivity and quality
  - c tools that increase project productivity and quality

When the two types of tools are coordinated, the total value of the resulting development support system is enhanced.

Many people have contributed to the state-of-the-art in programming technology. Acceptance by businessmen of a comprehensive set of consistent techniques followed practical demonstrations such as those reported by Harlan Mills and Terry Baker of the IBM Federal Systems Division. [35] Sources include a variety of books [22-24, 36-48] and other materials [49-57], discussing improved programming technologies (IPT) at a basic level. In 1976, IPT included:

- o structured programming
- a code reading
- o top-down development and step-wise refinement
- o structured design
- a pseudo-code program design language
- o BIPO
- o chief programmer teams
- o development support libraries
- o structured walkthroughs
- o inspections

The references contain a sample of IBM items readily available to this author. There are; of course, other sources for vendor manuals and reports.

Each year, some of these items will become so common in daily use that they will no longer be considered "improved" technologies.

New items will constantly be identified, proven by experiment, and added to the IPT list to maintain the stream of progress.

When these tools are all used on a project, the net result is expected to be an improvement in both the individual and the project aspects of productivity and quality. The reason is that the methods tend to build quality in from the beginning, eliminating the need for much of the costly scrap and rework that characterizes the tail end of many projects. Such results are "expected" but, unfortunately, not proven. It is improbable that a sound, repeatable experiment will ever be conducted to test the hypotheosis that IPT is better than old methods. Nevertheless, experienced managers are adopting IPT widely because, at a minimum, they find it decreases their risks. Without going into much detail, it is possible to see why.

- 2.3.2.1 Structured Programming (SP) vs. Unstructured Programming
  Managers are not comfortable managing products they cannot understand.

  They can understand structured code. They know that most discussion about programs takes place at the level of functions which are often recognizable as substructures of a good program. Therefore, SP makes it easier for a manager to talk to a programmer. SP makes it easier for a writer to document the program. SP makes it easier for a diagnostician to trace an error. SP makes it easier for a programmer to write the problem correctly in the first place.
- It is well know that the originator of a document is a poor proofreader of the result. For one thing, the writer retains enough of the material in his memory that that he sees what he expects to see rather than what is written. [Thus, the redundant "that" in the previous sentence is easily overlooked in proofing.] Retention is even more pronounced when copying data, particularly numbers. A keypuncher is so likely to make the same error on the second reading of a source document that it is normal to use a second person to verify the cards produced by the first. The same is true in publishing where an editor proofs the galleys prepared by a typist.

Proofreading by the buddy-system was also common in the early days of computing when programmers wanted help or wanted to show off good code. The practice ebbed as programming workload grew. Programmers no longer felt they could afford the time to help others. Code reading came back into style when it was pointed out that the practice (a) catches more errors than many other methods and (b) leads to better structure and more understandable programs.

A controlled experiment would require at least two independent developments of the same large scale system. Such an experiment is uneconomic. Furthermore, the normal variance in programmer skill can cause major productivity/ quality differences regardless of tools used by two independent groups. If one group does the job twice, the experiment also fails because the knowledge gained on the first version will dominate the factors contributing to productivity/quality in the second version.

The value of independent verification has been extended to the analysis and design activity in SADT. An integral part of Ross' method is the thorough review of each graphic document leading to an exchange of corrections, modifications, and suggestions between the reader and the author.

2.3.2.3 Top-Down vs Bottom-Up

Implementation from the top-down verifies each aspect of the design as it is developed. If the design is faulty, the faults can be dealt with immediately without propagating the effects to other parts of the system. Furthermore, the cost of getting the design right is minimized since there are only a small number of people involved during the design stage. Errors in a bottom-up implementation appear when code is complete at which time the maximum number of people are on board. Changes are handled in either approach by reiterating through the affected parts of the design. The results of making a change are more easily controlled in top-down design since all affected parts exist and no quesswork is required.

2.3.2.4 Structured Design vs Unstructured Design

Structured design refers to the conscious control of binding and coupling. Independent modules are truly independent and dependent modules have explicit interfaces and parameters. As a result, programmers can work on individual modules without fear of modifying other people's work.

Structured design also relies on explicit development of models of system data and system functions. Completeness can be checked by mapping the data models onto the function models. Maintenance is also improved by virtue of the clean interfaces which facilitate diagnosis and tend to pinpoint the source of a problem.

2.3.2.5 Program-Design Language vs Free-Form Design
One project that got into trouble was audited by a team of experts to see
what was required to complete the job. There were some fifteen people in
four groups working on what they said was a structured top-down design.
The auditors found each group using a different technique - flowcharts, PL/I,
prose, assembly-language - for design. No one knew what the others were
doing. Nor could the auditors figure out the design status. Here was a clear
case where the lack of a uniform design language resulted in a project failure.
The audit team recommendation was adopted: the project was taken away from the
fifteen people and reconstructed by another group at another location. The
procedural discipline of a program design language avoids such trouble.

2.3.2.6 HIPO VS Flowchart

The combination of functional data with procedural and managerial data gives everyone on a project a better view of what to do, when to do it, who is affected, and where each activity fits into the overall project. Simple flowcharts describe program structure but not project structure. Simple activity charts (PERT charts) show project structure but not program structure. HIPO or its equivalents show both.

2.3.2.7 Chief Programmer Teams vs Other Organizations

Team operations allocate tasks on the basis of skills and focus attention,
via the chief programmer, on top-down implementation. It is easier to use IPT
in a team organization. Communication is simpler, takes less time, and is
more effective in a team. The team is a hierarchy headed by the chief

programmer but, because it is small, personal relations can be excellent. Small teams can also operate effectively without a chief programmer because communications lines are short. Nevertheless, there is a lack of focus on the hierarchical nature of the program system making it somewhat more difficult to manage a top-down implementation. Large groups tend to lose the focus altogether and act as collections of peers with individual goals instead of a team with a single goal.

2.3.2.8 Development Support Libraries vs Private Code

When work in process is collected in an on-line library rather than kept in the programmer's desk several benefits appear. The official record of the system is accessible to all programmers and managers. Less time is spent handling card decks and tapes. Code reading is supported by current output listings. Interactive code, debug, and test aids can be installed. Skilled librarians can enter, maintain, and retrieve library records. They can also run tests as requested and distribute the output. Ultimately, delivery of the program package can be made automatically from the library.

A checkpoint is a date in a project schedule when an activity is to be completed and management intends to check that it is done. To avoid the problems caused when an activity is complete but wrong, the structured walkthrough converts the checkpoint into a careful examination of the completeness, accuracy, and general quality of the work product. The reviewers are often part of the development team. The programmer who did the work under review walks them through the program step-by-step, explaining how the program works and how the test plan will verify the program functions. A sample input is traced through the program. The reviewers then critique the work and recommend what, if anything, the programmer should correct, change, or restudy. Different aspects of the program draw attention at different stages of development as shown in Table 2.2. The whole process is quite informal. It relies on peer pressure from the reviewers on the developer to produce a high

quality product.

2.3.2.10 Inspections vs Structured Walkthroughs The success of a structured walkthrough depends on the personal relations within a project. Some walkthroughs are more successful than others. The more formal technique of inspection overcomes the variability of walkthroughs [46]. Inspections use a review team chaired by a moderator who organizes the review and reports the results. The team members are the designer, implementer, and tester of the program being inspected plus other technical experts or affected parties as necessary. Usually, four people are plenty. When an activity satisfies the project exit criteria for design completion or code completion, the moderator conducts an inspection. The inspection procedure follows a set pattern in which the design (at inspection  $I_1$ ) or the code (at inspection  $I_2$ ) is read and analyzed for logic and accuracy. Errors are identified (using a checklist to help). Based on the error rate, the moderator can either authorize the activity to proceed or send it back for rework (Fig. 2.12). Inspections are an explicit management tool. They capture quality data for management analysis and they result in management decisions affecting project schedules. Whereas walkthroughs were said to occur at checkpoints, inspections occur at milestones. The distinction between a checkpoint and a milestone is that the former verifies that an event occurred and the latter verifies the occurrence and requires a management decision to proceed or modify the project plan.

## 2.3.2.11 Support Tools

Much of the basic library of a program development organization provides support to the programmers. Compilers and assemblers support the decision to use a higher-level language. Trace programs, measurement aids, and utilities support program analysis, manipulation, and storage. Over and above these common tools are program systems designed solely to make programming easier and more reliable. The general characteristics of these support systems are:

- o interactive facilities both to help the programmer compose a program and to coach the programmer in location procedures and use of the tool
- o displays supplementing keyboard terminals to provide quiet access to a limited segment of a program or document plus the ability to access to a hard copy of the full listing when needed.
- central files of project control information, data set definitions, where used, tables, etc.
- d library services for storing programs, isolating work in process from released programs, and retrieving programs or program segments for authorized users.
- o text preparation services for editing programs and preparing system documentation.
- c test execution facilities based on a simple command system for selecting test cases and programs from the library and running a sequence of tests.
- d facilities for simulation and performance measurement
- o facilities for generating test cases
- o facilities to collect system measurements and user statistics for management reports.

Attempts to organize all these functions in a single system lead to extremely large support facilities which become development problems themselves. Therefore, it has been recognized that the best support system is a collection of individual tools which have been designed to work together within a common architecture. It should be possible to link the tools together automatically to produce a useful sequence of operations such as program entry, compile, test case generation, link edit, load and execute, file program and tests for future use, prepare and release results.

By planning the approach around separate tools, the full capability of a support system can be obtained by a development shop which cannot afford to build its own. Pomeroy [58] listed some of the tools that were commercially available in 1972. Since then the list has expanded to include various on-line systems, display processors, structured programming aids, syntax checkers, data base managers, text processors, test generators, command systems, and reporting aids [59]. The major drawback is that these tools are seldom designed to a consistent architecture even within a single vendor's catalog. On the other hand, it is not hard to modify the output of one to satisfy the input requirements of another.

An organization with many programmers simultaneously coding, debugging, testing, and shipping programs will have one or more computers dedicated to the support system. Most organizations will have to use their operational computer. In either case, the preferred way to protect one activity from another when they share a single CPU is to use a virtual machine operating system. Such a system creates an artificial machine environment for each user. As a result, the operational version of an installation OS can be isolated from the version used by developers. This is particularly helpful when the OS itself is being modified.

A great deal of the benefit from a support system comes from its library capabilities. It is the repository of all the project data. All project personnel can be given access to project data to learn it, to use it, to work on it, or to analyze it for design or management reasons. Nothing gets lost in a programmer's desk. "Private" code becomes "public" code. All the communications links needed to move the project forward can be provided through the support system. These capabilities appear in various degrees in the large CLEAR/CASTER system used in IBM program development [60], the planned Information Automat [61], ISDOS under development at the University of Michigan [62], and in the minicomputer operating system from Bell Laboratories, UNIX [63].

## 2.3.3 Applicability of Technical Tools

Development and test tools, particularly structured programming, are better established than analysis/design tools. The reasons for this are based on the characteristics of the program development life cycle discussed in the next chapter. Briefly, the most obvious place to look for the source of program errors or cost overruns is to the individual programmers. When this was done in the late 1960s it became immediately apparent that the large number of people who entered the field since 1960 were not doing basically "good" programming. In the '40s and '50s programmers carried a task from start to finish by themselves. They did structured programming and stepwise refinement without thinking about it. It was the natural way to work. But, because they had not thought about techniques, they did not document or teach the methodology for "good" programming. In the '60s, individual jobs were fewer (except for the special case of personal computing) and team jobs failed to give individuals a context in which to develop "good" habits. Techniques such as structured programming were rediscovered by theoreticians who, fortunately, explained their ideas in a teachable format. Then, when the team programmers learned how to distinguish "good" programs from "bad" ones, the largest part of the quality and cost problems was solved. Or so it seemed. In fact, the progress made with IPT served to highlight the fact that good programmers cannot compensate for poor designs. This discovery led to the development of improved design methodologies which could interface with and take advantage of IPT. Still there were serious problems, now due to the frequent changes and misunderstandings arising from poor requirements statements. So in the '70s, work on requirements analysis techniques accelerated.

### 2.3.3.1 Scope of Selected Tools

Tools developed for specific purposes tend to be efficient only within the limits of their design even though they may be used elsewhere. By analogy, FORTRAN fits well in mathematical problems but is clumsy as a commercial data processing language. REVS which was built to help requirements writers is not promoted as a design tool. SADT is offered to

cover both analysis and design but does not claim to be a programming tool at the implementation level. HIPO exhibits a different type of specialization. It addresses only documentation and does not have the functional sophistication of the design approaches. HIPO, however, requires practically no investment by the user. Self-teaching is quite feasible. It does not do much but what it does is important, of wide use, and economical. So far, simplicity and economy coupled with formal rules and standard procedures characterize IPT tools. Other analysis/design/development/test tools tend to be more sophisticated. They give the user freedom of action to express his own style in the "art" of analysis and design. The price of such flexibility is that they cost more to acquire and use and generally require a training course to be used properly.

None of these tools is necessarily important to a programmer working totally alone, particularly, when he is the only person that will ever use the program. The tools pay off best in team projects, either medium size projects (Fig. 2.13) where analysis and high-level design may be merged or large size projects (Fig. 2.14) where more specialization of skills is found. Looking at the scope of just a few tools (Fig. 2.15), you could conclude that REVS is of greatest interest in large projects where requirements analysis is a special skill separate from design. IPT fits all cases. Further, the stress on good documentation and control techniques lets all the tools contribute to better maintenance and other follow-up activities.

## 2.3.3.2 Benefits of IPT

Efforts to put a price tag on the benefits of IPT have been largely unsuccessful because of the lack of controlled experiments. It is equally risky to draw conclusions from isolated examples where productivity and/or quality improved simultaneously with the introduction of IPT. Analysis of such cases often shows other factors also contributed to the improvement. The principal other factor is the increase in the average level of experience of the staff over the period of the sample. For instance, a trend observed in United States data for the period 1974-76 will reflect the fact that there was little hiring or employee turnover among programmers for those years. The same people were being observed for three years. Even though there is no proof that experience improves productivity and quality it is highly probable that if an improvement is observed it is as much due to experience as anything else. The most believable data would have to come from very large programming organizations (over 300 programmers) which have many jobs in process and have converted some but not all to IPT. In this case, the numbers would be large enough to allow averaging over aggregates of individuals and jobs. The mix of characteristics of all the IPT jobs can be assumed to be the same as the mix for all the non-IPT jobs. Now, if a trend is observed in production rates, error rates, or resource utilization rates it could be attributed to IPT.

There is no such clear cut experimental analysis to this author's knowledge. Very good analyses, however, are attributable to Fagan [51] and Jones [9]. Both had access to extensive production and maintenance records on numerous program systems; however, the data that had been collected was not complete nor was it consistently defined. Therefore, many of the analytical results represent the investigator's insight and consensus opinions. Fagan observed the programming locations of IBM in Kingston and Poughkeepsie, New York; Jones observed a smaller group at IBM San Jose but augmented his data from other sources. Fagan predicts that structured programs undergoing design and code inspections will have substantially fewer errors per KLOC than structured programs without formal inspections. His measurements show that

inspections I, and I are the key to the saving. They catch errors early and avoid error propagation. Both inspections occur before any significant machine time is used; that is, before program unit debugging starts. Fagan also finds a 20% saving in the coding portion of development. Jones, working from a different data base and a planning model, compared IPT projects with other methods: an "old style" in which program unit debugging was the earliest quality control, an "OS/360 style" in which design verification was also done, a "structured style" combining design verification with top-down design and structured programming, and a "modern style" where inspections supplement the structured style. Jones also found about 40% reduction in errors/KLOC when comparing OS/360 style to modern style. Savings in post-release service are projected by Jones ranging from 40-95% of the four year cost of fixing errors in program products with many users. Considering that in four years, service of OS/360 style programs uses one-third as many programmer-years and one-and-a-third as many computer hours as the total initial development of the type of products Jones is concerned with, IPT savings represent big money.

The effects of four different quality control approaches are shown in Table 2.3. Using Jones' old style as a base, the errors found in a project are allocated to the development cycle stages: design, code, debug, test, and post-release. Thus, in old style no errors are found by explicit quality controls until the debugging stage. There, 15% of the errors are found; 60% are found in system integration and test; 25% of the errors remain for users to find. By contrast, most of the errors are found in the design and code stages when the Modern Style is followed. The table entries for Old Style and OS/360 Style are interpreted from Jones' data. The Structured Style is an average of data from several sources where IPT was used but inspections were not used because the project team preferred structured walkthroughs or even less rigorous reviews. Modern Style is an average of Jones, Fagan, and other sources which reflects a range of optimism allocating from 25-55% of the errors found to the design stage.

An important feature of the various techniques is the cascade effect of early error detection. An error eliminated during design tends to forestall additional errors later. As a result, the total number of errors encountered in a modern style project is less than in an old style project. Some experts in large system design predict that IPT eventually will be ten times better than old style. Here, a more modest improvement of 45% is projected based on what has been achieved to date.

It is possible to see the dramatic value of IPT by using the costs of fixing an error detected in a given stage (Table 2.4) to estimate the relative costs of each quality control approach. Remember that the cost of an error is proportional not only to the difficulty of finding a solution (fix) but also to the number of people whose progress is stalled while waiting for the fix. During design there are only a few people involved and they know their product intimately. During test there are many people involved who, at best, know only a small part of the product. After release, the errors are reported by users who submit incorrect and duplicate reports to maintenance teams who may be quite unfamiliar with the nuances of the product design. These factors account for the huge increase in cost per fix

at later stages. If you were to translate the relative factors to dollars, you would immediately see the benefit of cleaning up errors early. Using the values in Tables 2.3 and 2.4, Table 2.5 presents the relative cost per stage and the total cost per project of quality control. The details are carried out to decimal values to be consistent with Tables 2.3 and 2.4. The totals are rounded off since the original data, while taken from actual environments, is not precise. Still, there is a 7:1 improvement in quality projected over the period 1960-1976. If the trend is consistently maintained the vendor's costs will continue to go down but the main beneficiary will be the user who will have fewer problems. For every 25 errors that reached the customer under the Old Style, only 2-4 should reach him today and still fewer in the future. This speaks well for the quality control aspect of program development but it does not address other issues of productivity and cost.

Using the same data as Fagan and Jones, Horne in early 1976 set out to show the effectiveness of IPT in terms that non-programming managers would understand. This objective would require statistically valid results large enough to be convincing. This objective was not achieved for a variety of reasons; the most important of which was that the source data had not been obtained for the purpose of this study. Most of the data was used by local managers to track departmental performance. Horne needed to draw general parameters out of data that was collected for local, specialized purposes. She had collected a considerable amount of data on IBM system control programs (e.g., MVS, VS1) and application programs (e.g., IMS, health industry packages, government and commercial contract programs). In this list were very large complex programs, widely distributed programs, small simple programs, one-user programs. The data included varying degrees of detail on program size, size of changes to existing programs, IPT used, programmer-months, estimated difficulty, productivity, errors found before and after release, and other items. However, the data was unreliable. Measurements were not consistent and were not well defined. For example, one program appeared in four reports each of which gave different size, productivity, cost, and error rates per MACC. "Productivity" was sometimes limited to KLOC/programmer-month and in other cases covered KLOC/project personnel-months but it was hard to tell which was which. The IPT usage reports were often subjective. You could not tell if a project reporting IPT usage understood the techniques claimed. For instance, inspections are different from walkthroughs but few people were aware of the difference. The applications people did not report cost or post-release errors. Reports for new programs were intermixed with reports for new releases of old programs. In fact, in the total KLOC in the programs surveyed there were far more KLOC of old code than there were new or changed KLOC. And, to complicate matters, the individuals who collected the data were no longer around so it was difficult to clarify issues of definition and scope.

Nevertheless, in the analysis by Horne and this author it was possible to confirm the belief that IPT improves the program development process based on the following indications (as opposed to proofs):

- For projects of comparable difficulty, IPT increased productivity in KLOC/people-years on the average.
- Managers were willing to commit to higher productivity on new projects increasing the KLOC/programmer-month used in their estimating tables.

- 3. Managers were willing to commit to lower objectives in terms of errors/KLOC in post-release code because they had confidence in their quality control techniques.
- 4. Managers were convinced that IPT helped them and they were voluntarily adopting the techniques.
  - 5. Programmers were willing to adopt new techniques in 1976 whereas they shied away from them in 1974-75.

On the negative side, the indications were:

- It is so far not possible to separate the benefits of one technique from another.
- Unusually high productivity (found in a few 1-5 programmer projects) is due more to outstanding programmers than to IPT.
- 3. Determining the effect of IPT (or any other process modification) on quality probably requires about two years of post-release error data. However, when trends have been established for several programs, it should be possible to model the results to predict quality in other programs given only pre-release error data.
- 4. For every set of data supporting IPT, there is a counterexample.
- 5. The range of results with IPT overlaps the range without. Since the data is statistically weak, no conclusion can be drawn about the average results due to IPT.
- 6. If IPT data solidifies so improvement in the averages can be seen, the effect on process measurements may remain invisible because the savings can be masked by other factors.
  - o the cost of system support, integration, and release may dwarf the cost of new code so IPT benefits appear negligible.
  - o the savings due to IPT may be immediately reinvested in additional program functions; i.e., total outlay is not reduced.
  - o the reduction in errors/KLOC may be accompanied by an increase in KLOC so the user sees no change in practice.
  - o the savings in programmer-hours may simply reduce programmer overtime without changing salary costs or programmer population.

An interesting sidelight of the Horne study was due to an attempt to fit curves to the data. The purpose was to be able to predict costs and error rates by class of program, difficulty, and workload. The form of the curves that best fit the data was

k (A) (A A) + a

where A is the size in KLOC (or modules) of the program systems, A A is the number of new or changed KLOC (or modules), k is a factor related to programmer productivity, and a is a constant related to design quality and process management. The equation reflects the discussion of Chapter 1 by representing complexity as  $A^2$  and workload as AA/A and their product as (A) (ΔA). Fitting this expression to error data yields values of k and a. The variable portion, k, can be interpreted as the number of errors delivered in a program system due to programmer activity. The constant portion, a, which should be zero is the number of delivered errors due to the development process. The surprise was that in this model when k = 0 (i.e, each programmer did a perfect job of program unit coding and debugging), a was non-zero. The process itself was introducing a certain number of defects. If the analysis is correct, these system errors must be due to (a) inadequate design, (b) errors put in after unit debugging, and (c) the nature of test and release activities. IPT is expected to improve design and minimize the test and release activities. Support tools that hold the system on-line can reduce handling errors after unit debugging. Therefore, the new technology may get a down to zero where it belongs.

A different approach to evaluating IPT was taken by Hunter and Reed [64] who subjectively rated the use of IPT on a large project at Barclays Bank Ltd. in the United Kingdom. They found significant but unquantified gains in product quality, enhanced personnel skills and motivation, and about 25% better productivity than on other jobs with which they were familiar. The techniques used wholly or in part were:

- o structured design
- o pseudo-code program design language
- .a structured programming
- o top-down implementation and testing
- o development support library
- o librarians
- o walkthroughs
- o team operations.

The project received advice and assistance from the group who wrote the IPT manuals [50]. As a result, the joint Barclay-IBM team was able to get off to a fast start. At the end of the project, a set of questions was prepared to show where IPT made a difference. The strong positive bias of the answers showed that IPT had been a success in this particular project. A summary of the questions covering manageability, productivity, maintainability, and reliability shows how the qualitative assessment turned out [Table 2.6].

The conclusions of this study parallel those given earlier:

- o IPT provides a framework for project development which is better than the traditional approach and should become the standard.
- o The techniques are not a 'miracle formula' for a successful project. Without a sensible schedule, strong technical leadership, and competent design and programming, a project will fail with and without IPT.
- o Ideally, projects should introduce most of the techniques together, starting as early in development as possible. The techniques fit naturally together and in some cases depend for their success on being used in conjunction with others.
- o However, a subset of the techniques can easily be used as a starting point. It should consist of structured programming (including pseudo-code), use of a development support library approach, and walkthroughs.
- o Standards in many cases need to be altered to accommodate development along IPT lines.
- o The overhead cost of learning the techniques is almost certainly recovered inside one year.
- o IPT is not a rigid or inflexible set of rules. Rather it is a group of related concepts and techniques. It is natural and inevitable for different projects to use and interpret IPT somewhat differently.

Taking advantage of the plusses of IPT, it is likely that modern style program development will lead to quality and productivity improvements. IPT may be hard to measure; however, the improvements due to IPT over time will be measurable. By 1980, IPT should generate about 40% improvement of the old style of 1970 [65]. That is, 40% fewer errors found during development and more than a 40% reduction in errors in code released to users (because of the cumulative effect of early discovery). Project resource requirements per KLOC should also go down about 40% represented by an increase in KLOC/programmer-month and in KLOC/computer-hour. The range of 1-4 KLOC/programmer-month (before adjusting for system support, system test, release, and maintenance as discussed in Chapter 10) is achievable. This outlook is within the range of the technology forecast for 1985 made by the Air Force Systems Command in its 1972 study "Information Processing/Data Automation Implications of Air Force Command and Control Requirements in the 1980s (CCIP-85)" reported by Boehm and Kosy [66]. Their range is very large due to the inherent variability of programming. It forecasts productivity of 3.7 KLOC/programmer-month in 1980 rising to 5.3 KLOC/pm in 1985 compared to 1.4 KLOC/pm in The increase is attributed to IPT but includes programming languages, design aids, and reuse of common functions (macros/library programs).

These numbers are the midpoints of ranges estimated from a logarithmic scale in Boehm. Boehm's chart shows 0.4-7 KLOC/pm in 1980, 0.6-10 KLOC/pm in 1985, and 0.2-2.6 KLOC/pm in 1970. Kosy has a similar chart with a slightly different scale which yields a more optimistic range in 1980 of 1.4-15.5 KLOC/pm. The lower range is probably more realistic.

The accompanying text is part of a larger chapter covering system characteristics and status control tools as well as support tools. The entire reference list is included here since there are some useful items in it.

# References - Chapter 2

- [1] ACM Forum, CACM, Vol. 18, No. 11, November 1975. The State of Computer Methodology: Gorn, S., "When The Chips are Down" and Schwartz, J. T., "What Constitutes Progress in Programming?"
- [ 2] Baram, M. S., "Technology Assessment and Social Control", Science, Vol. 180, May 4, 1973.
- [ 3] Herzberg, F., Work and The Nature of Man. Cleveland, Ohio: World Publishing Co., 1966
- [4] Levinson, H., The Exceptional Executive: A Psychological Conception.

  Cambridge, Mass.: Harvard University Press, 1968. Also Newman, W. H.,

  Administrative Action. Englewood Cliffs, N. J.: Prentice Hall, Inc., 1951.
- [5] Opler, A., "Fourth Generation Software", DATAMATION, Vol. 13, No. 1, January 1967.
- [6] Aron, J. D., "Commerical Data Processing Machines in Government Applications", AFIPS Conference Proceedings, Vol. 40, Spring Joint Computer Conference 1972. Montvale, N. J.: AFIPS Press, 1972.
- [7] Banham, J. A. and P. McClelland, "Design Features of a Real Time Check Clearing System", IBM Systems Journal, Vol. 11, No. 4, 1972. Also DATAMATION, Vol. 22, No. 7, 1976 Special Feature on Electronic Banking and Computer Decisions, Vol. 8, No. 3, 1976 Features on Electronic Funds Transfer (EFT).
- [8] Weiler, P.W., R. S. Kopp, and R. G. Dorman, "A Real-Time Operating System for Manned Spaceflight", IEEE Transactions on Computers, Vol. C-19 No. 5, May 1970.
- [ 9] Jones C., "Program Quality and Programmer Productivity," TR 02.764. San Jose, Cal:IBM General Products Division, 1977.
- [10] Boehm, B. W., "Software Engineering", IEEE Transactions on Computers, C-25-12, December 1976.
- [11] Kawamura, K. and A. N. Christakis, "Methods for Structural Modeling".

  Proceedings of the IEEE Conference on Cybernetics and Society.

  New York: N.Y.: IEEE, Inc. 1976.

- [12] Forrester, J. W., Industrial Dynamics. Boston, Mass.: The MIT Press and New York, N.Y.: John Wiley & Sons, Inc., 1961.
- [13] Teichroew, D. and E. A. Hershey III, "PSL/PSA: A Computer-aided Technique for Structured Documentation and Analysis of Information Processing Systems".

  IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, January 1977.
- [14] Bell, T. E., D. C. Bixler, and M. E. Dyer, "An Extendable Approach to Computer-Aided Software Requirements Engineering." Also, Alford, M., "A Requirements Engineering Methodology for Real-Time Processing Requirements." And Davis, G. C., and C. R. Vick, "The Software Development System". IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, January 1977.
- [15] Ross, D. T., and K. Schoman, "Structured Analysis for Requirements Definition".

  Second International Conference on Software Engineering, October 1976. IEEE

  Transactions on Software Engineering, Vol. SE-3, No. 1, January 1977. Also
  Ross, D. T. and J. W. Brackett, "An Approach to Structured Analysis",

  Computer Decisions, Vol. 8 No. 9, 1976.
- [16] Jackson, M. A., Principles of Program Design. London, England: John Wiley & Son Ltd, 1970.
- [17] Langefors, B., Theoretical Analysis of Information Systems. Lund, Sweden: Studentlitteratur, 1966. Also Langefors, B., and B. Sundgren, Information Systems Architecture. New York, N. Y.: Petrocelli/Charter, 1975.
- [18] Warnier, J. D., Logical Construction of Programs. Leiden, Netherlands:
  H. F. Stenfurt Kroese, B. V., 1974. Also New York, N.Y.: Van Nostrand-Reinhold, 1976.
- [19] Schwartz, J. T., "Principles of Specification Language Design with Some Observations Concerning the Utility of Specification Languages". in Algorithm Specification. Rustin, R. ed. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1972.
- [20] DeRemer, F. and H. H. Kron, "Programming-in-the-Large Versus Programming-in-the-Small", IEEE Transactions on Software Engineering. Vol SE-2, No. 2, June 1976.

- [21] Dijkstra, E. W., "Notes on Structured Programming." The Report 70-WSK-03, Eindhoven, Netherlands: Technical Hochschule, April 1970.
- [22] Dahl, O.-J., E. W. Dijkstra, and C. A. R. Hoare, Structured Programming. New York, N. Y.: Academic Press, 1972.
- [23] Wirth, N., Systematic Programming: An Introduction. Englewood Cliffs, N. Y.: Prentice-Hall, Inc. 1973.
- [24] Mills, H. D., and R. H. Linger, Structured Programming and Software Engineering. Reading, Mass.: Addison-Wesley Publishing Co., 1977.
- [25] Van Leer, P., "Top-Down Development Using a Program Design Language".

  IBM Systems Journal, Vol. 15, No. 2, 1976.
- [26] Hamilton, M. and S. Zeldin, "Higher Order Software A Methodology for Defining Software". IEEE Transactions on Software Engineering, Vol SE-2, No. 1, March 1976.
- London, R. L., "The Current State of Proving Programs Correct", Proceedings of the ACM Annual Conference, August 1972. Also, Elspas, B., K. N. Levitt, R. J. Waldinger, and A. Waksman, "An Assessment of Techniques for Proving Program Correctness", ACM Computing Surveys, Vol. 4 No. 2, 1972. Also Owicki, S. and D. Gries, "Verifying Properties of Parallel Programs: An Axiomatic Approach", CACM, Vol. 19, No. 5, 1976.
- [28] HIPO A Design and Documentation Technique Installation Management Series, Form GC20-1851. White Plains, N. Y.: IBM Corp., 1974. Also Katzan, H., Systems Design and Documentation: An Introduction to the HIPO Method. New York, N. Y.: Van Nostrand Reinhold Co., 1976.
- [29] Barr, L., V. LaBolle, and N. E. Willmorth. Planning Guide for Computer Program Development. Report No. TM-2314, Santa Monica, Calif: System Development Corp., 1965.
- [30] Jones, N. M., "HIPO for Developing Specifications", DATAMATION, Vol. 22, No. 3, 1976. Also Stay, J. F., "HIPO and Integrated Program Design,"

  IBM Systems Journal, Vol. 15, No. 2, 1976.

- [31] Guest, J., S. Lai, and R. L. Loyear, "Simulation Technique of a Distributed Intelligence System", 1975 Summer Computer Simulation Conference. LaJolla, Cal.:Simulation Councils, Inc. 1975.
- [32] Benwell, N. ed, Benchmarking-Computer Evaluation and Measurement. Washington, D.C. Hemisphere Publishing Corp. 1975. Distributed by John Wiley & Sons, New York.
- [33] Wilkes, N. V., "Software Engineering and Structured Programming." IEEE
  Transactions on Software Engineering, Vol. SE-2, November, December 1976.
- [34] Lucas, H. C., Jr., "Synthetic Program Specifications for Performance Evaluation," Proceedings of the ACM Annual Conference, 1972 New York, N.Y.: Association for Computing Machinery, 1972.
- [35] Mills, H. D., "Software Development". IEEE Transactions on Software Engineering Vol. SE-2 No. 4, December 1976. Also, Mills, H.D., "Top-down Programming in Large Systems," in Debugging Techniques in Large Systems, R. Rustin, editor. Englewood Cliffs, N.J.: Prentice-Hall, 1971. Also Mills, H.D., "Chief Programmer Team: Principles and Procedures", Report FSC 71-5108. Gaithersburg, Md.:IBM Federal Systems Division, 1971. Also, Mills, H. D., "Mathematical Foundations for Structured Programming", Report FSC 72-6012. Gaithersburg, Md:IBM Federal Systems Division, 1972. Also, Mills, H. D., "How to Write Correct Programs and Know It", Report FSC 73-5008, Gaithersburg, Md.: IBM Federal Systems Division, 1973. Also, Mills, H. D., "On the Development of Large, Reliable Programs," Proceedings of the IEEE Symposium on Computer Reliability. New York, N.Y.: IEEE, 1973. Also, Mills, H. D., and F. T. Baker, "Chief Programmer Teams: DATAMATION, Vol. 19, NO. 12, 1973. Also, Baker, F. T., "Chief Programmer Team Management of Production", IBM Systems Journal, Vol. 11, No. 1, 1972. Also, Baker, F. T., "System Quality Through Structured Programming-Programming", AFIPS Conference Proceedings, Vol. 41, Fall Joint Computer Conference 1972. Montvale, N.J.: AFIPS Press, 1972. Also, Baker, F. T., "Structured Programming in a Production Environment". IEEE Transactions on Software Engineering. Vol. SE-1 No.2, June 1975.
- [36] Aron, J. D., The Program Development Process Part 1- The Individual Programmer. Reading, Mass.: Addison-Wesley Publishing Co., 1974.
- [37] Brooks, F. P., Jr., The Mythical Man-Month. Reading, Mass.: Addison-Wesley Publishing Co., 1975.

[38] Conway, R. and D. Gries, An Introduction to Programming. Cambridge, Mass.: Winthrop, 1973. [39] Dijkstra, E. W., A Discipline of Programming. Englewood Cliffs, N.J.: Prentice-Hall, 1976. [40] Kernighan, B. W. and P. J. Plauger, The Elements of Programming Style. New York, N. Y.: McGraw-Hill, 1974. [41] McCracken, D. D., A Simplified Guide to Structured COBOL Programming. New York, N.Y.: John Wiley & Sons, Inc., 1976. [42] McGowan, C.L. and J. R. Kelly, Top-Down Structured Programming Techniques. New York, N.Y.: Petrocelli/Charter, 1975. Myers, G. J., Reliable Software Through Composite Design. New York, [43] N.Y.: Petrocelli/Charter, 1975. [44] Weinberg, G. M., The Psychology of Computer Programming. New York, N.Y.: Van Nostrand Reinhold, 1971. Yourdon, E. and L. L. Constantine, Structured Design, New York, N. Y.: Yourdon, [45] Inc., 1975. Also, Yourdon, E., How to Manage Structured Programming. New York, N.Y .: Yourdon, Inc., 1976. Yourdon, E., Techniques of Program Structure and Design. Englewood Cliffs, [46] N.J.:Prentice-Hall, 1975. [47] Special Issue: Programming. Articles by Brown, P. J., Yohe, J. M., Wirth, N., Khuth, D. E., Kernighan, B. W. and P. J. Plauger, edited by Denning, P.J. ACM Computing Surveys, Vol. 6, No.5., 1974. Symposium on Structured Programming COBOL - Future and Present. New York, N.Y.: [48] Association for Computing Machinery, 1975. [49] IBM Installation Management Series. White Plains, N. Y. , An Introduction to Structured Programming in COBOL, GC20-1776, 1975. , An Introduction to Structured Programming in PL/I, GC20-1777, 1975. , HIPO - A Design and Documentation Technique, GC20-1851, 1974. , OS Development Support Libraries, GC20-1663.

, Improved Programming Technologies - An Overview, GC20-185.

- [50] IBM World Trade Corporation. Improved Programming Technologies Series.

  Copehagen, Denmark.

  , Management Overview, GE19-5086, 1975.

  Partch, H. B., The Programming Dilemma: Maintenance vs Development.

  GE19-5085, 1975.

  , Code Reading, Structured Walk-Throughs, and Inspections.

  GE19-5200, 1976.
- [51] Fagan, M. E., "Design and Code Inspection to Reduce Errors in Program Development", IBM Systems Journal, Vol. 15, No. 3, 1976.
- [52] Greaves, T. J., Learning to Use Structured Programming or, How to Think in a Structured Manner. TR54.043, IBM Corp., White Plains, N. Y., 1974.
- [53] Opdyke, H. G., An Approach to Team Programming. TROO.2613, IBM Corp., White Plains, N. Y., 1975.
- [54] Stevens, W. P., G. J. Myers, and L. L. Constantine, "Structured Design," IBM Systems Journal, Vol. 13, No. 2, 1974.
- [55] Thomas, D. J., A Definition of Top-Down Programming System Development TR00.2562, IBM Corp., White Plains, N. Y., 1974.
- [56] Waldstein, N. S., The Walk-Thru, A Method of Specification, Design and Code Review. TR00.2536, IBM Corp., White Plains, N.Y., 1974.
- [57] Waldstein, N. S. and R. J. Alulis. The Library Controller. TR00.2633, IBM Corp., White Plains, N. Y., 1975.
- [58] Pomeroy, J. W., "A Guide to Programming Tools and Techniques", IBM Systems Journal, Vol. 11, No. 3, 1972.
- [59] Software Directory. Vol. 1-Data Processing Management. Vol. 2- Business Applications. Carmel, Indiana:International Computer Programs, Inc. Published in January and July.
- [60] Brown, H. M., "Support Software for Large Systems", in Buxton, J. N. and B. Randell (eds.), Software Engineering Techniques. Brussels, Belgium: NATO Scientific Affairs Division, 1970.
- [61] Mills, H. D. and M. L. Wilson, "An Introduction to the Information Automat", SHARE European Association Proceedings, Anniversary Meeting 1975. Also "The Information Automat Approach to Design and Implementation of Computer Based Systems." IA Report, 1975. Available from IBM Federal Systems Division, Gaithersburg, Maryland.

- [62] Teichroew, D., "Information Processing Systems Analysis and Design."

  Progress Report C00-25-44-1. Michigan University, Ann Arbor, Michigan,

  January 1 September 30, 1975.
- Ritchie, D. M. and K. Thompson, "The UNIX Time-Sharing System", CACM, Vol. 17, No. 7, 1974. Also, Rochkind, M. J., "The Source Code Control System". Proceedings of the 1st National Conference on Software Engineering. New York, N. Y.: IEEE, Inc., 1975 and IEEE Transactions on Software Engineering, Vol SE-1 No. 4, December 1975.
- [64] Hunter, I. C., and J. N. Reed, An IPT Project-Results, Conclusions, and Recommendations. , 1976.
- [65] INFOTECH, Structured Programming Survey reported in Financial Times, March 14, 1977.
- Boehm, B. W., "Software and Its Impact: A Quantitative Assessment".

  DATAMATION, Vol. 19 No. 5, 1973. Also Kosy, D. W., Air Force Command and Control Information Processing in the 1980s: Trends in Software Technology. Report R-1012-PR. Santa Monica, California: Rand Corp., June 1974.
- [67] PERT: Guide for Management Use. PERT Coordinating Group. Washington, D. C.: Superintendent of Documents, U. S. Government Printing Office, 1963.
- [68] PERT Cost. DOD and NASA Guide, Washington, D. C.:Office of the Secretary of Defense and National Aeronautics and Space Administration, June, 1962.
- [69] Moder, J. J. and C. R. Phillips, Project Management with CPM and Pert. New York, N.Y.: Van Nostrand Reinhold Co., 1970.
- [70] Ackoff, R. L. and N. W. Sasieni, Fundamentals of Operations Research. New York, N.Y.: John Wiley & Sons, 1968.
- [71] Clark, W., The Gantt Chart. London: Sir Isaac Pitman and Sons, 1938.
- [72] Kernighan, B. W. and P. J. Plauger, <u>Software Tools</u>. Reading, Mass: Addison-Wesley Publishing Co., 1976.

PROBLEM STATEMENT

OBJECTIVES

REQUIREMENTS

ARCHITECTURE

HIGH-LEVEL DESIGN

DETAILED DESIGN, CODE, DEBUG

INTEGRATION, TEST, RELEASE

OPERATION, MAINTENANCE, UPGRADE

TERMINATION

TABLE 2.1 PROGRAM PROJECT STEPS

		Project checkpoints	Items to be reviewed via a structure walk-through					
		End of system planning	Project plans System definition Task identification					
nts		Major project review 'technical'	Functional specifications Work assignments Schedules					
project checkpoints	oints	Detailed Design	Internal specifications HIPO package					
project	r checkp	Coding	Uncompiled source listings					
Major	Multiple minor checkpoints	Documentation	User guides Programmer maintenance manuals o Internal specifications o HIPO package					
		End of Development	Deliverable product o Code o Documentation					

TABLE 2.2 STRUCTURED WALKTHROUGH AT PROJECT CHECKPOINTS (from [45.3]

Project Quality Stage Control Approach	Design (% err	Code ors found	Debug	Test	Post Release	System Errors (relative to old STYLE
OLD STYLE debug test	0	0	15	60	25	100
OS/360 STYLE  design review  debug,test	20	0	15	50	. 15	98
STRUCTURED STYLE  IPT with  walkthroughs	25	20	15	35	5	58
MODERN STYLE  IPT with  inspections	45	25	7	20	3	55

TABLE 2.3 QUALITY CONTROL - % OF ERRORS FOUND BY FOUR QUALITY CONTROL APPROACHES

STAGE	RELATIVE COST
DESIGN	0.3
CODE	0.5
DEBUG	2.5
TEST	21.0
POST-RELEASE	36.0

TABLE 2.4 QUALITY CONTROL - RELATIVE COST OF FIXING AN ERROR FOUND IN EACH DEVELOPMENT CYCLE STAGE

PROJECT STAGE QUALITY CONTROL APPROACH	DESIGN (Cost	CODE	DEBUG OLD STYLE	TEST	POST RELEASE	TOTAL PROJECT (rounded value)
OLD STYLE	0	0	37.5	1260.0	900.0	2200
OS/360 STYLE	5.88	0	36.75	1029.0	529.2	1600
STRUCTURED STYLE	4.35	5.8	21.75	426.3	104.4	560
MODERN STYLE	7.425	6.875	9.625	231.0	59.4	315

cost entries are product of errors per stage times cost from tables 2.2 and 2.3

TABLE 2.5 QUALITY CONTROL - RELATIVE COST OF FIXING ALL ERRORS BY STAGE AND BY TOTAL PROJECT

		YES	TO IPT	No	2	EFF	ECT	
MANA	GEABILITY							
0	Are major schedules being met more often?	(1)	( , )	(	)	(		
0	Are budgets being met more often?	(1)	( )	(	)	(	)	
0	Are milestones and checkpoints easier to set up and to enforce?	(1	(1	(	)	(	)	
0	Are intermediate project milestone and checkpoints being met more often?	s ( 🖍	1	(	)	(	)	
0	Do project managers get better sta reports to track progress?	tus	(1)	(	)	(	)	
o	Is the developing system more visible, centralized, and open to inspection?	(1)	<b>(/</b> )	(	)	(	)	
0	Is the user more involved with the system development effort?	(1)	(1)	(	)	(	)	
0	Can changes be made to the developing system more easily?	(1)	1	(	)	(	)	
0	Can part of the system be exercise and shown to the user before the e of the project?		(/)	(	)	(	)	
o	Are project managers more confiden about their ability to manage and their job satisfaction higher?	is (	1	(	)	(	)	
0	Are analysts and programmers more satisfied with their jobs?	(1)	( )	(	)	(	)	
٠.	Can people be phased into projects more easily?	( )	( )	(	)	( -	5	
0 -	Are clerical tasks easier to delegate?	15	(1	(	)	(	)	
0	Is the user more satisfied?	(5	(	(	)	(	)	
•	Is the chief executive more satisfied?	(5	1	(	)	(	)	
0	Is the DP manager more satisfied?	(1)	1	(	)	(	)	

TABLE 2.6 Subjective Evaluation of IPT

Sheet 1

		YES	TO IPT	N	07	EFF	ECT
PROI	DUCTIVITY						
•	Is there less confusion between users and DP?	( )	( )	(	)	( )	1
0	Is the identification of responsibilities more sharply defined?	( )	( )	(	)	( *	1
0	Can analysts and programmers find what they need to know more easily	?()	(	(	)	(	)
0	Can programmers schedule their time more effectively?	(1	()	(	)	(	)
0	Does the system need fewer drivers than similar systems produced in the past?	(1)	1	(	)	(	. )
٥	Is there less throw-away code?	1	(1)	(	)	(	)
0	Is the amount of machine test time less?	(1	(-	(	)	(	)
o	Can machine time now be scheduled and used more effectively?	()	(	(	)	(	)
0	Is machine turnaround time less of a productivity bottleneck?	(1	(N)	(	)	(	)
0	Can test data and test cases be generated more systematically?	( )	( )	(	)	( ,	7
0	Does it seem that DP is producing more output for a unit of time?	1	( )	(	)	(	)
•	Does it seem that DP is producing more output for a unit of cost?	6	( )	(	)	(	)

TABLE 2.6 Subjective Evaluation of IPT

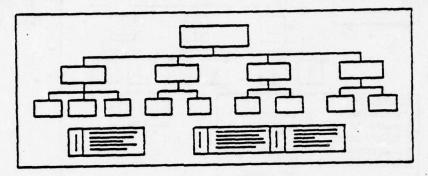
Sheet 2

		YES	YES DUE	NC	2		NO	
MAI	NTAINABILITY & RELIABILITY							
•	Does the developing system seem to have fewer errors?	(/)	(1)	(	)	(	)	
•	Does it take less time and effort to fix the errors that do occur?	(1)	1	(	)	(	)	
o 	Is it taking less machine time to find and fix errors?	(1)	1	(	)	(	)	
0	Are the errors that occur less catastrophic than in the past?	( )	( )	( ,	7	(	)	
0	Is integration test less traumatic than in the past?	(	(1)	(	)	(	)	
0	Once into operation, does the syst have fewer errors?		(1	(	)	(	)	
٥.	Does it take less time to make a maintenance change?	15	(V)	(	)	(	)	
0	Is the documentation more meaningful on this system?	(1)	(1)	(	)	(	)	
0	Is the documentation easier to update?	(5	(	(	)	(	)	
0	Do programmers show less reluctance to maintain someone else's code?	e ( • )	(	(	)	(	)	
•	Is there less of a stigma surrounding maintenance?	( )	( )	(	)	( -	7	
•	Are fewer programmers required to maintain this system than similar systems?	6	6	(	)	(	)	

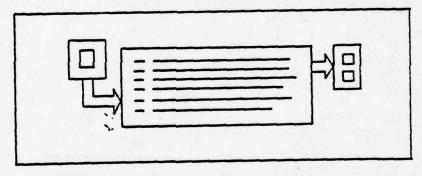
TABLE 2.6 Subjective Evaluation of IPT

Sheet 3

Visual Table of Contents



Overview Diagrams



Detail Diagrams

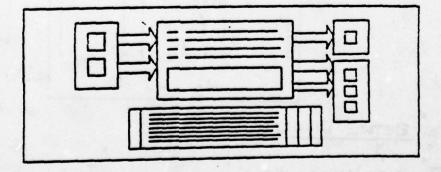
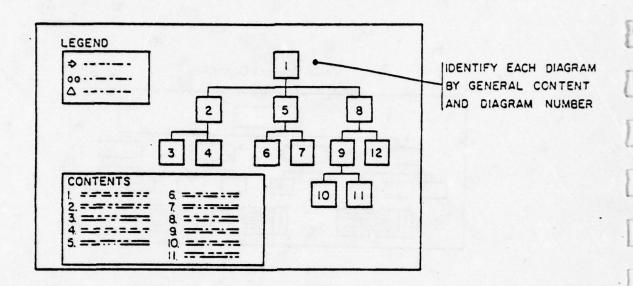
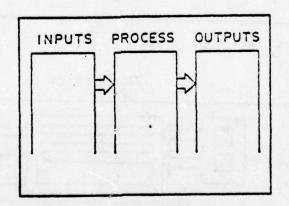


FIG. 2.8 HIPO TOP-DOWN STRUCTURE



a. HIERARCHY AND INDEX



b. DETAIL DIRERAM

Fig. 2.9 HIPO CHARTS

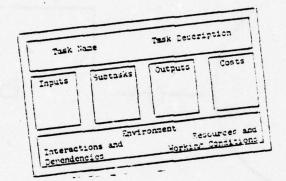
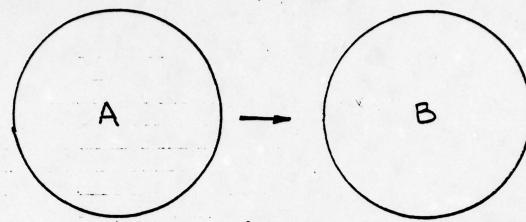
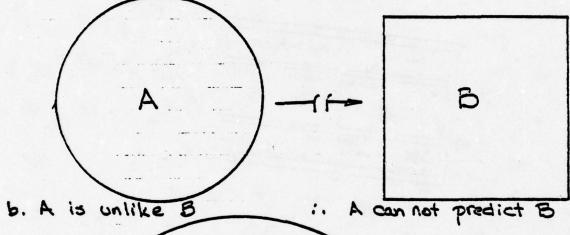
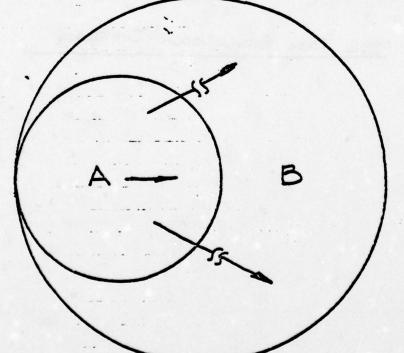


FIG. 2.18 SDC TASK BREAKDOWN DIAGRAM



A is like B in size and function : A can predict B





e. A 1s like B but smaller i. A underestimates B

FIG. 2.11 PREDICTABILITY USING A REFERENCE SYSTEM

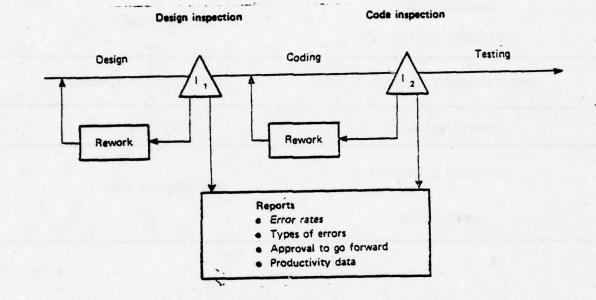


FIG. 2.12 INSPECTIONS (from [45.3])

	<b>*</b>
PROBLEM STATEMENT	USER
OBJECTIVES	+
	<b>A</b>
REQUIREMENTS	SYSTEM ANALYSIS
ARCHITECTURE	&
	DESIGN TEAM
HIGH-LEVEL DESIGN	4
	<b>A</b>
DETAILED DESIGN, CODE, & DEBUG	INDIVIDUAL PROGRAMMER   ✓
	<u> </u>
TEST & RELEASE	TEST TEAM
	<b>^</b>
MAINTAIN & UPGRADE	MAINTENANCE TEAM
TERMINATION	

FIG. 2.13 ASSIGNMENTS IN MEDIUM SIZE PROJECTS

	<b></b>
PROBLEM STATEMENT	USER
OBJECTIVES	CONSULTANT .
REQUIREMENTS	SYSTEMS ANALYSTS
ARCHITECTURE	SYSTEMS ARCHITECTS
HIGH-LEVEL DESIGN	SYSTEM DESIGNERS
DETAILED DESIGN, CODE, & DEBUG	INDIVIDUAL PROGRAMMERS
TEST & RELEASE	TEST TEAM
MAINTAIN & UPGRADE TERMINATION	MAINTENANCE TEAM

FIG. 2.14 ASSIGNMENTS IN LARGE PROJECTS

PROBLEM STATEMENT

OBJECTIVES

REVS

REQUIREMENTS

JACKSON
WARNIER

HIGH - LEVEL DESIGN

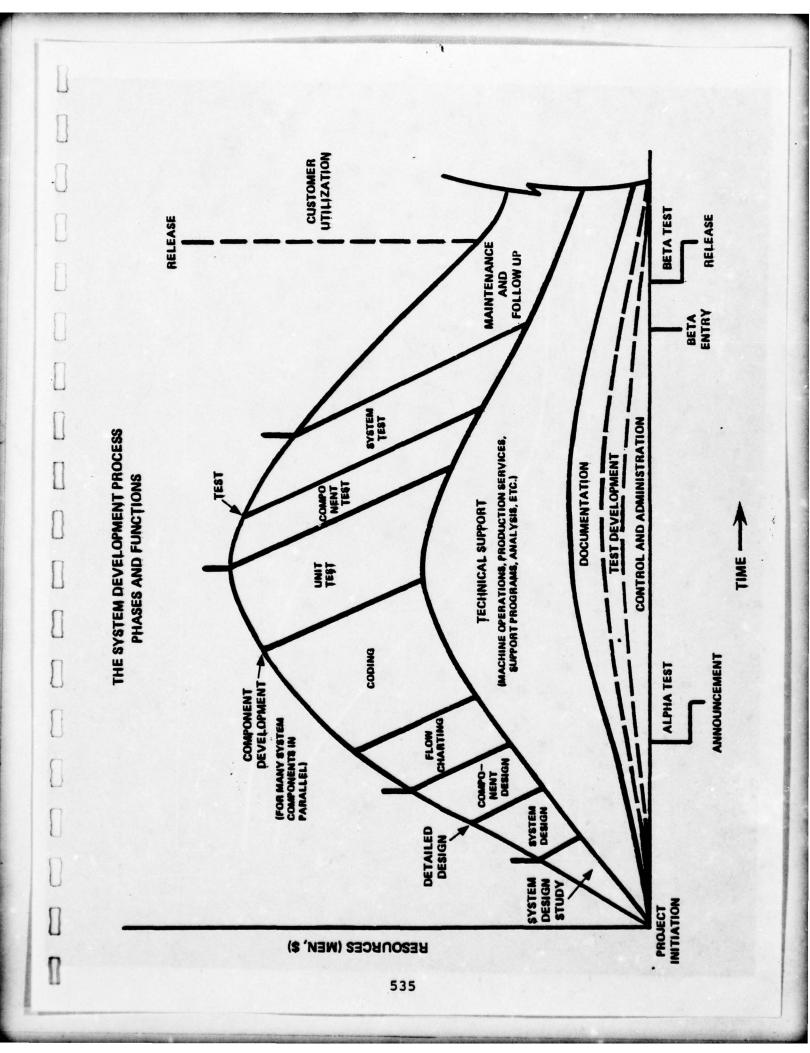
DETAILED DESIGN, CODE, & DEBUG

TEST & RELEASE

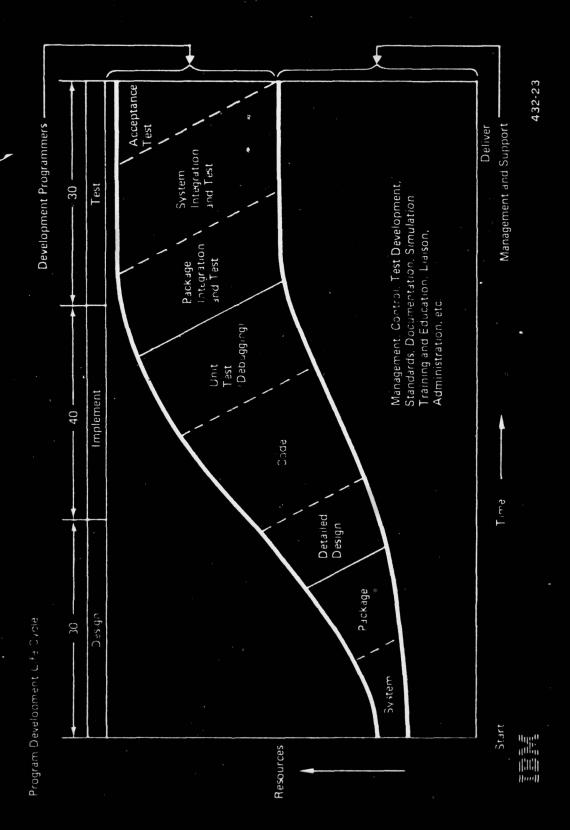
MAINTAIN & UPGRADE

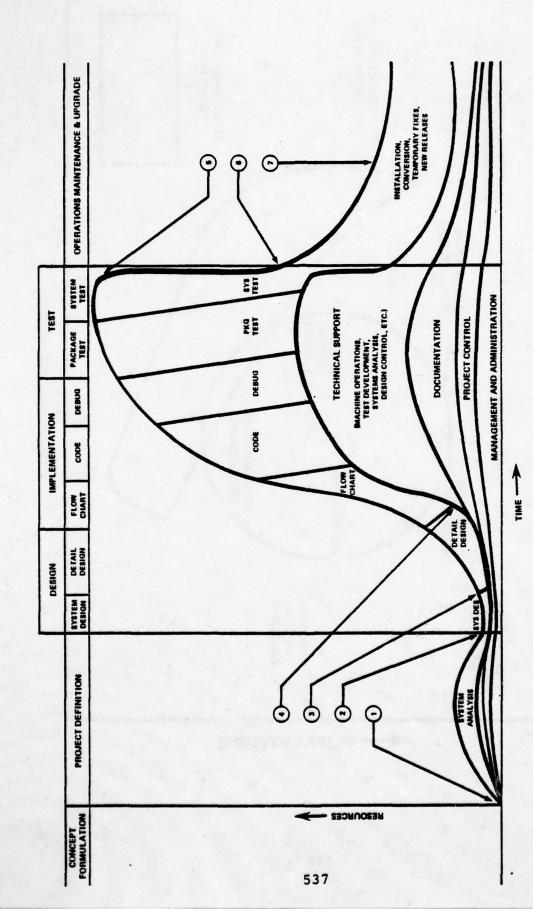
TERMINATION

FIG. 2.15 SCOPE OF SELECTED TOOLS



# SOFTWARE MANAGEMENT

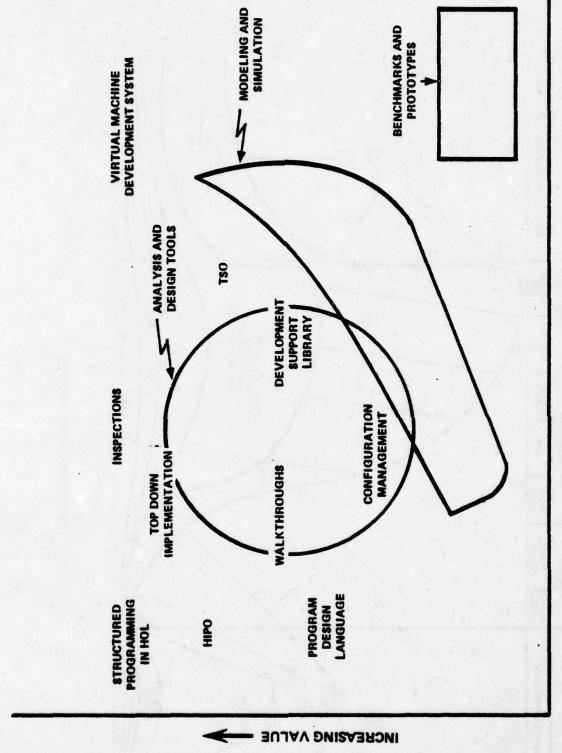




Townson of the last

THE SYSTEM DEVELOPMENT LIFE CYCLE

-,-



INCREASING COST -

Commen

U

#### APPLYING SADT TO LARGE SYSTEM PROBLEMS

by .

John W. Brackett Clement L. McGowan

SofTech, Inc.

#### ABSTRACT

SADT is a language for describing systems. It encourages multiple descriptions of a system from different viewpoints. We sketch its basic characteristics stressing the natural synergy between a language and the thought patterns it induces. SADT has been successfully applied to many diverse systems to perform requirements definition and functional analysis. It has also been used for both software design and for planning in non-computer related areas. We detail actual approaches used in applying SADT that integrally involve customer analysts. Some representative examples of SADT projects are discussed, stressing their distinctive features.

#### 1. WHAT IS SADT?

SADT . Structured Analysis and Design Technique, is a language for describing systems together with a methodology for producing such descriptions. Here a "system" may be defined as any combination of machinery (hardware), data, and people working together to perform a useful function. More generally a system can be viewed as consisting of things (objects, documents, or data), happenings (activities performed by people, machines, software, etc.), and their interrelationships. This fundamental starting point accounts for SADT's broad applicability. For example, the definition of requirements (without necessarily specifying whether they are to be achieved by people, by software, by machines), a structured software design, and even a p3 organization (people, papers, and procedures) are "systems" and thus amenable to a precise SADT description.

SADT was developed initially by Douglas T. Ross during the period 1969-1973. It was drawn from his own ongoing work in problem solving and in software engineering dating back to the late 1950's. It was strongly influenced by ideas from Hori's human-directed activity cell model [2], structured programming and its supporting procedures, software engineering, general systems theory, and even to some extent cybernetics.

SADT has been further developed and refined at SofTech as a result of extensive use dating from 1973. It has thus far been successfully applied to a variety of complex system problems ranging from real-time telephonic communications design and computer-aided manufacturing to the analysis of funds management and of military training.

#### 1.1 THE NOTATION AND ITS MODES OF THOUGHT

An SADT system description is a set of diagrams each depicting only a limited amount of detail. The diagrams expose a system structure a piece at a time from the top down. Quite literally SADT espouses the principle that "to divide is to conquer", provided we know how the divided pieces are combined to constitute the whole. This proviso applies to the diagram contents as well as to the relationships among diagrams. The rules of language usage encourage and even enforce unfolding a system description in intellectually manageable information units. At all times the system structure and hence the relationship of any part to the whole remains graphically visible.

As a language SADT has a syntax, a semantics, and, of course, an associated pragmatics of use. Its syntactic notation has an elegant simplicity that has been introduced in several publications [4, 5, 6, 11]. Diagrams consist principally of boxes (representing parts of a whole) interconnected by arrows (representing interfaces between the parts) each suitably labelled with nouns (for things) or verb phrases (for happenings).

Each box on a diagram (representing, say, a system activity) can be further detailed on a separate diagram with more interconnected boxes and arrows. The notation requires that a box and the corresponding diagram detailing it represent <a href="mailto:exactly">exactly</a> the same part of a system. In particular, [SADT is a trademark of SofTech, Inc.]

the external interfaces in the form of inputs, outputs, and controls must match ("the geometry goes together"). Thus, the diagrams in an SADT system description can be placed directly into a tree-like hierarchy. This information is encoded by an indexing scheme in a natural fashion so that the appropriate context for any diagram can be immediately determined. Figure 1 depicts two diagrams. A box on the first diagram is detailed by the second diagram.

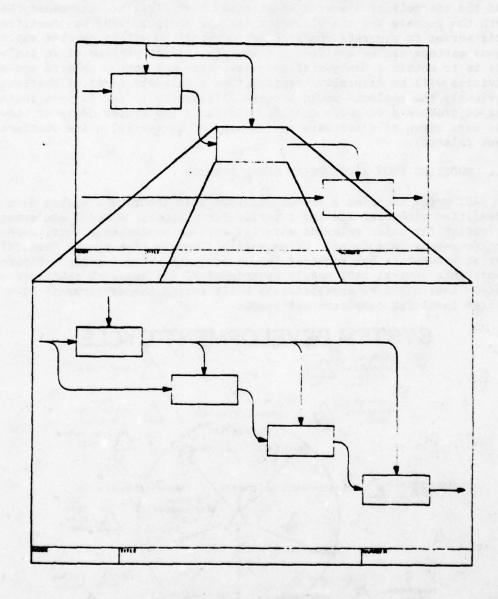


Figure 1: A diagram detailing the box of another

Bertrand Russell has written that "a good notation has a subtlety and suggestiveness which at times makes it seem almost like a live teacher .... a perfect notation would be a substitute for thought." SADT as a

language and as a methodology directs and disciplines the analysis of systems. Crucial concepts are naturally expressible and thus are more easily discerned and then communicated. For instance, system feedback and control can be concisely and clearly captured. It is indeed "a good notation."

As with any language, ease of expression encourages certain patterns of thought. Within SADT one is naturally led by the notation to initially bound the context of consideration. That is, the inputs, the outputs, and the controls of a system under study must first be determined. Next, both the purpose and the viewpoint for the analysis must be identified. This serves to separate concerns by focusing the effort so that extraneous matters can be avoided. For example, if the purpose of an analysis is to obtain a management overview, then the system objects and activities will be naturally examined from a suitable level of abstraction. Obviously the analysis would proceed differently if the purpose instead was to produce a software design. Similarly the system designer view and the user view of a software system are, of necessity, quite distinct (but related).

#### 1.2 MODELING FROM A VIEWPOINT FOR A PURPOSE

An SADT model contains a set of diagrams that describe a system from an identified viewpoint and for a particular purpose. A model can provide a context for other relevant material such as explanatory text, supporting documents, and forms. Often multiple models of a system from differing viewpoints are required for an adequate understanding. Figure 2 represents several SADT models by triangles. It suggests the range of models that might be appropriately built during the development of a system involving computers and people.

# SYSTEM DEVELOPMENT CYCLE

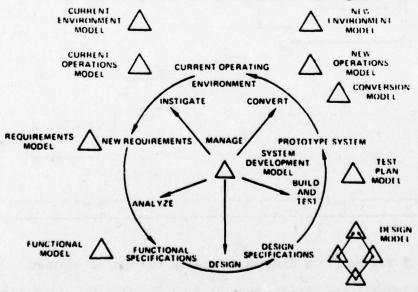


Figure 2: Models appropriate to System Development Cycle Stages

The range extends from conception through installation and even includes the management of this process. The scope of certain key models should be mentioned. A Requirements Model crystallizes criteria the new system must meet in order to satisfy the requestor's needs (along with those of the manager, users, etc.). A Functional Model expresses what functions the new system must perform and their implications. A Design Model depicts a particular design that meets the requirements by performing the specified functions. Actually the design model in Figure 2 is shown as an interconnected group of models. This represents SADT's capability of isolating important design decisions for separate treatment as models.

Figure 3 shows two models (i.e., system descriptions with different view-points) sharing common detail. Consistency can be checked by "tying" the different models for a system together, i.e., by making explicit their interrelationships.

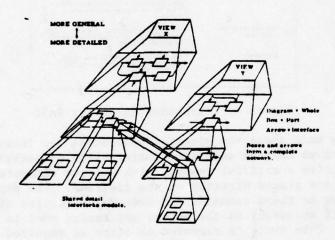


Figure 3: Two interconnected models of a system from different viewpoints

# 1.3 P3 - THE PEOPLE, THE PAPERS, AND THE PROCEDURES

Producing an SADT system description is a demanding technical project requiring management, expertise, and regular reviews to assess current status. Clearly extensive interaction with the customer along with "interviewing" appropriate documents is a necessary step in systems analysis. SADT models and procedures provide a methodology to structure this mass of information. Building models to reflect different purposes and viewpoints effectively sorts and relates the system details.

Drawing inspiration from the recent innovative organization of a software project into a team of specialists, SADT has defined functions for roles identified as important in obtaining a simple, clear, correct, and complete system description. Figure 4 lists job titles and associated duties.

Hame	Zuacuen.		
Authore	Personnel who etudy requirements and constraints, analyse system functions and represent them by models based on SADT diagrams.		
Commontero	Usually authors, who must review and semment in writing on the work of other authors.		
Readers	Personnel who read SADT diagrams for information but are not expected to make written comments		
Esperte	Persons from whom authors obtain specialized information about require- ments and constraints by means of interviews.		
Technical Committee	A group of sonior technical personnel assigned to review the analysis at every major level of decomposition. They other receive technical issues or recommend a decision to the project management.		
Project Librarian	A person assigned the responsibility of maimeining a centralized file of all project documents, making cepies, distributing reader this, keeping records, etc.		
Project Manager	The member of the project who has the final technical responsibility for carrying out the eyetem analysis and design.		
Monitor for Chief Analyst)	A person fluent in SADT who accists an advices project personnel in the use and application of SADT.		
Instructor	A person fluent in SADT, who trains Authors and Commences using SADT for the first time.		

Figure 4: Personnel Roles for SADT

Much could be said about the specific procedures for interaction and reporting. Here we mention only the crucial cycle of authoring diagrams that then receive a critical reading with written comments. These reader veactions are placed directly on the diagrams. The author must react in writing to these comments and eventually revised diagrams are issued. Only if necessary do the author and reader meet to discuss any differences. This cycle is repeated as often as required to evolve a complete and correct model. If the readership is suitably selected, this process effectively ensures the clarity as well as the accuracy of the resulting system descriptions. In effect it institutionalizes an "egoless" system analysis process.

#### 2. APPLYING SADT

In this section we indicate the diversity of areas to which SADT has been applied. We comment on some common characteristics. The actual approaches to applying SADT in terms of personnel roles and the interaction with SofTech are sketched. Then some specific examples of SADT projects are discussed.

#### 2.1 AREAS OF APPLICATION

The most important characteristic of SADT application projects has been a need to communicate results among people with different interests and backgrounds who must be kept informed during the development of a large manual or automated system. The SADT development process is based on the fact that analysis and design of complex systems requires coordination

of the creation, modification, and verification of the system description. No single person can comprehend completely every aspect of a complex system within the time limits usually imposed. Although SADT has been profitably used on one- or two-person projects, a typical project involves several persons; the largest project to date has involved fifteen analysts simultaneously working on different aspects of the requirements for a large financial information system. Nearly all of these projects have required frequent communication of the results from the analysts producing a system description to experts in areas impacted by the system and to management responsible for utilizing the system.

The applications to which SADT has been applied are sufficiently diverse that at first glance no pattern appears to exist that would indicate where it might be most useful. The largest use to date has been performing requirements definition and functional analysis in areas where computer systems are to be developed. However, SADT has also been used to:

- Describe aerospace manufacturing operations to sufficient detail to provide the basis of a long-range plan for computer-aided manufacturing.
- Design a data base system to maintain and modify the records of over five million business establishments participating in Federal Unemployment Insurance programs in sufficient detail to produce module design specifications using high-level structured PL/I.
- Describe how the peace-time Army provides a combat-ready tank system in order to identify the training required and the circumstances in which training must be carried out to keep a weapon system ready.
- Document the current budgeting and accounting operations of a large government agency in order to isolate the areas which could benefit the most from improved information support.

The pattern linking most SADT applications appears to be the difficulty of the problem to be solved. All the problems listed above were very hard, and to solve them required the analysts and designers to approach the solution in a highly-structured way and to work as a team with effective division and coordination of effort. SADT contributed in both areas and allowed personnel who were not particularly knowledgeable about many aspects of the area being studied to synthesize the information obtained from many experts and to develop a solution whose adequacy and quality was assured by regular, critical review within the development team and with potential users and non-technical personnel.

#### 2.2 APPROACHES TO APPLICATION

Since a key part of SADT is a structured way of thinking about large and complex problems, it is best learned through application under the supervision of a person experienced in its use. Serving an "apprenticeship" is the preferable approach; training is not possible solely through class-

room or self-paced instruction. The establishment of the review process requires both management involvement and the training of commenters to read SADT models and to document their suggestions directly onto copies of the author's diagrams. Project managers must be coached to take advantage of the increased visibility of the analysis and design process in order to better evaluate project status and to improve estimating, pacing, and resource allocation.

The normal approach to innovating SADT in an organization has been to conduct a technology transfer program for the personnel assigned to a single project (typically 5-10 persons). Following a two-week course in the basics of the methodology, the project team begins to work on its problem "under the wing" of an experienced SADT practitioner, the monitor. The monitor assists and advises project personnel in the use and application of SADT and teaches seminars on advanced topics as they become relevant to the work of the project. The monitor trains commenters, ensures the review cycle is working properly, and trains the SADT librarian. In addition the monitor frequently provides consulting to the project manager on the most effective use of SADT and the evolving models, and reviews the quality of the on-going work.

A three-month monitoring period has usually been adequate to effectively transfer technical aspects of the methodology and to develop the disciplined team approach that provides a large fraction of the observed SADT benefits. It is desirable for the project to remain in contact with the monitor by telephone. The monitor should also make a one-day visit every two or three weeks in order to review quality and to help the project personnel stand back and take a better look at their problems. This approach has been used by SofTech in over twenty technology transfer programs in the last three years.

An alternative to the monitoring approach when more trained SADT personnel are available is to use a combination of trained personnel and learners on the project, with the most experienced personnel monitoring the work of less experienced people. However, a monitor who is not directly involved in doing analysis or design can more effectively review quality and provide consulting to the project manager. An example of using experienced and inexperienced personnel together is a project to study current operations and to develop the requirements for a financial information system where five SofTech experienced SADT authors are working closely with ten client analysts both to complete the project and to train the client personnel in SADT.

SADT also facilitates analysis and/or design of a system by an outside organization in order to fulfill user requirements. For such a working relationship to be effective, continuous and effective communication is required throughout the project since the analysts and designers assigned to the project are not likely to be particularly knowledgeable about the users' areas of expertise. SofTech's experience has been that an evolving SADT model with its supporting text and references to other documentation provides the understandable, complete, uniform and current documentation such a working relationship requires. The documentation must show analysis or design decisions in context so that they can be challenged

while alternative approaches are still viable. Such documentation must provide the reasons for decisions and facilitate review by user experts. Attention must be given early to establishing the review cycle and to defining how analysis and design decisions are to be critiqued. In all cases where these steps were taken SofTech has had excellent results in solving a wide variety of user problems using professional analysts whose generic analysis skills are enhanced by SADT.

In the following section examples of the three approaches described above are given in the context of several recent applications of SADT.

#### 2.3 EXAMPLES OF APPLICATION

This subsection describes some recent SADT projects concerned with soft-ware design, current financial operations, computer-aided manufacturing, and training of personnel. They are representative of the spectrum of areas in which SADT has been successfully applied to large systems problems. For each project its goal as well as the actual working relation-ship with the customer are outlined. The project output in terms of models and number of diagrams are also presented. In some cases proprietary rights permit only a generic description of the project and the contracting customer.

#### 2.3.1 PABX Software Design

This completed SADT project was to produce detailed software design for a microprocessor-based private automatic branch exchange (PABX) telephone system. It had to accommodate a wide range of installation size (100 lines up to 10,000 lines) and over 275 special features. The functional specifications were given using a program design language specially tailored for specifying switching programs.

Three customer authors and one SofTech author built two principal SADT models — a design model and an implementation model. In all about eight models and approximately 500 diagrams were fashioned. The extreme quantity of diagrams can be attributed to the detail to which the analysis was taken. Boxes at the bottom level sometimes reflected individual statements in a PL/I-like language.

In many instances the diagrams were used as programmer work assignments. The use of SADT coupled with structured programming practices led to an increase of between 2 and 3 times the productivity previously attained by the customer's programmers. An outside contractor was also used for system implementation and was able to easily read the diagrams in order to produce code.

The building of multiple models from different viewpoints (e.g., operator, designer, user, and manager viewpoints) was a major factor in this highly successful software design effort. It served to isolate the important system interfaces that were then accommodated by the design. The central role of SADT diagrams as self-contained communication documents was repeatedly underscored.

## 2.3.2 Financial Management

This SADT project is describing the current operations of a large federal government agency's financial management system for the purpose of identifying needs and deficiencies. After construction of a current operations model, a requirements model for a financial management information system is to be built. As many as 15 authors have been working simultaneously at four geographically dispersed sites. Customer participation in the modeling effort is substantial with a 3 to 1 ration of its authors to SofTech authors.

Three basic models are being constructed (with other supporting or auxiliary models as needed): Obtain Financial Resources, Manage Funding, and Accounting. About 150 diagrams plus many pertinent documents and forms will comprise the final financial management system description. The eventual goal is to design and then implement management information systems that are coordinated and integrated in order to handle the anticipated dynamic growth in the agency's financial information requirements. Design models may be built to realize this goal.

The PSL/PSA system [9, 10] is being used to record the activity, data, hierarchy, and structure expressed in the SADT models. PSA produces reports that summarize this information. The system detects naming conflicts (as a consistency check) and also guards against proliferating synonyms.

Periodic model walk-throughs using project personnel were a crucial complement to the machine checking and tracking of data usage. Judiciously invoked, the walk-throughs have proven to be a major device to keep the project manager cognizant of the current status while exercising the dynamic sequencing inherent in the models.

Several facets make this SADT project distinctive: the number of authors involved at multiple sites, the fact that they are preponderately customer analysts, and the use of PSL/PSA as a computer support tool for the SADT methodology.

# 2.3.3 Computer-Aided Manufacturing

As part of the U. S. Air Force Integrated Computer-Aided Manufacturing (ICAM) program SofTech is using SADT to develop a manufacturing "architecture" — a generally applicable model of the batch manufacturing process, with initial emphasis upon sheet metal fabrication and subassembly. The SADT Architectural Model will show the hierarchical structure of all aspects including machines, materials, processors, procedures, people, and organizations. It will describe the essentials of batch manufacturing that are common across companies in order to plan subsequent steps of the ICAM program.

The project will utilize SADT to capture the manufacturing experience of the aerospace and industrial concerns participating in the project (viz., Boeing, General Dynamics, Grumman, Hughes Aircraft, Lockheed, Rockwell International, United Technologies Corporation, Vought, Caterpillar

Tractor, Cincinnati-Milacron, Giddings and Lewis, and Keaney and Trecker). Multiple models of different aspects of manufacturing are being developed by SofTech analysts based both on interviews with experts at the participating companies and on documents provided by them. Extensive review of the evolving models will be performed at each company by company experts especially trained to serve as the interface between their manufacturing experts and the SofTech SADT authors.

The PSL/PSA system [9, 10] is being used to store the evolving models and to perform a variety of consistency audits. Computer support to SADT was deemed necessary since many diagrams will be created in the multiple models representing manufacturing from different viewpoints.

SofTech is also conducting major manufacturing studies in the areas of advanced processing planning and geometric modeling for Computer-Aided Manufacturing-International. In both programs, large manufacturers are working closely with SofTech using SADT models as the main communication vehicle.

### 2.3.4 Army Training

SADT is being used by SofTech in conjunction with the Headquarters of the U. S. Army Training and Doctrine Command (TRADOC) to:

- Identify changes in Army training required to significantly increase combat effectiveness.
- Describe how Army training, testing, and evaluation programs will operate after the proposed changes are accomplished.
- Plan how the changes in Army training will be undertaken and how progress will be monitored and evaluated.

The project is being jointly funded by TRADOC and the Defense Advanced Research Projects Agency (DARPA) in order to demonstrate the use of SADT in the context of the Army's current effort to make significant improvements in training. An important project objective is to assess whether better systems analysis methods will facilitate the definition of military training as a large system. SADT models are being developed to show the Army training system in terms of clearly described activities and their interfaces. Such a system definition should provide the basis for more effectively solving many small problems, which must be handled with limited resources and under tight deadlines, in the context of overall military objectives.

The project includes an independent evaluation by a team from the University of Texas under the direction of Dr. Gary Borich which will provide DARPA with:

- An assessment of the utility and effectiveness of SADT in the training command environment.
- An evaluation of the impact of the project on TRADOC's capabilities to define and manage innovations in Army training.

A recently completed project task has been identifying changes in training support to increase the combat readiness of major Army weapons systems. The tank was selected as a prototype for purposes of identifying the training support needed. An SADT model describing the tank as a total weapon system [8] was used to determine requisite training changes. The model consists of 38 diagrams with supporting text. It was constructed by a team of two SofTech authors and two Army authors over a period of four months.

The process of applying SADT included interviewing many Army experts and obtaining comments from 25 Army commentators at TRADOC Headquarters and at the Armor School, Fort Knox.

This project is different from those described above in that project goal is not the definition or the development of a computer-based system, but the demonstration of how SADT can be used for analysis and planning in non-computer related areas.

### 3. THE SIGNIFICANCE OF SADT

We take the applicability of SADT to large system problems as an established fact. As a language for describing systems, SADT solves the simultaneous need for precision and for clarity of communication. Its notation is rigorous without being unduly formal, for a system description must encompass myriad technical details while being accessible to a readership concerned principally with an overview. The use of multiple descriptions each reflecting a particular purpose and viewpoint represents a major advance in organizing our understanding of systems.

SADT may point the way to the eventual impact that language theory, and in particular semantics, will have on large real world systems. In a sense, SADT provides a semantic framework in terms of which all analyses and descriptions are done. The variety of successful applications serves as a pragmatic existence proof for the adequacy of the SADT framework.

The Reference section reflects the relatively recent emergence of SADT-related literature in the public domain. We close by quoting Dr. Michael Melich [3] of the Center for Naval Analysis on using SADT:

"With my interests in performance measurement of command and control systems, the ability to partition functions unambiguously gives me a big start on the development of performance measures. I now have a way to get people to agree about what they think one of these systems is supposed to do. Once they agree, I have a chance to decide if the systems actually do what they are supposed to do. The last and probably one of the most surprising consequences of this use of Structured Analysis has been the rather severe reduction in time wasted needlessiy arguing. It seems that we actually can define and stick to the point without useless wandering on 'sea stories' or ' I remember whens'."

#### REFERENCES

- 1. Air Force Materials Laboratory, AFSC, WPAFB, Air Force Computer-Aided Manufacturing (AFCAM) Master Plan (Vol. II, App. A, and Vol. III) (Report #AFML-TR-74-104 available from DDC as AD922-041L and 922-171L), July 1974.
- Hori, S., Human-directed activity cell model. CAM-I long range planning final report, <u>CAM-I</u>, Inc., 1972.
- 3. Melich, M., Evolution of naval missions and the naval tactical data system. Proc. Conf. on Managing the Development of Weapon System Software, 1976, pp. 26-1 26-36.
- Ross, D. T., Structured Analysis: A language for communicating ideas. <u>IEEE Trans. on Software Engineering</u>, 3, 1, 1977, pp. 16-34.
- Ross, D. T. and Brackett, J. W., An approach to structured analysis. <u>Computer Decisions</u> 8, 9, 1976, pp. 40-44.
- Ross, D. T. and Schoman, K.E., Structured analysis for requirements definition. <u>IEEE Trans. on Software Engineering</u>, 3, 1, 1977, pp. 6-15.
- SofTech, Inc., TRAIDEX Needs and Implementation Study (Final Report)
   DARPA Contract #MDA 903-75-C-0224, May 1976. (available through
   NTIS AD-A024 861).
- SofTech, Inc., Task 2 Report: the role of training in providing the combat-ready tank system. DARPA Contract #MDA-903-76-C-0249, October 1976 (to be available through NTIS).
- 9. Teichroew, D. and Hershey, E. A., PSL/PSA -- a computer aided technique for structured documentation and analysis of information in processing systems. <u>IEEE Trans. on Software Engineering</u>, 3, 1, 1977, pp. 41-48.
- Teichroew, D. and Sayani, H., Automation of system building. <u>DATA-MATION</u> 17, 16, 1971, pp. 25-30.
- 11. Thomas, M., Function decomposition: SADT. Proc. Infotech Conf. on Structured Design, 1976, pp. 115-135.

# DOMONIC - A SYSTEM TO DOCUMENT, MONITOR, AND CONTROL SOFTWARE DEVELOPMENT

# SOFTWARE LIFE CYCLE MANAGEMENT (SLCM) WORKSHOP Sponsored by

U.S. Army Computer Systems Command
Warrenton, Virginia

August 22-23, 1977

by

Dick B. Simmons
Texas A&M University
College Station, Texas 77843

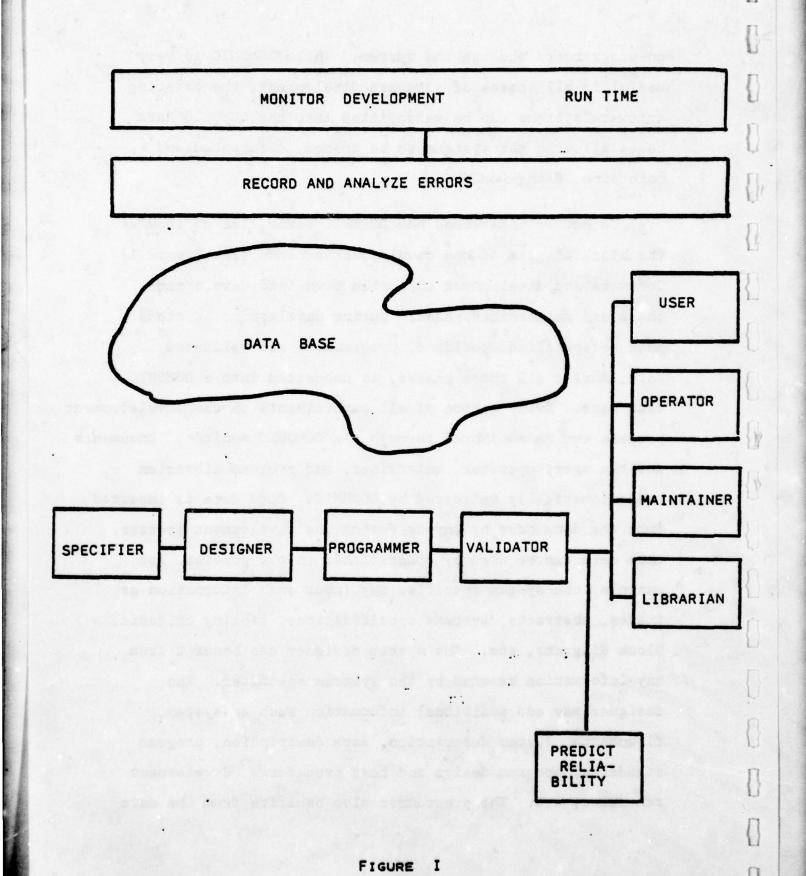
## INTRODUCTION

During 1972, Texas A&M University made a study of documentation aids for NASA at Goddard Space Flight Center. The results of this study led to the design and development of DOMONIC (A System to Document, Monitor and Control Software Development). From the early stages of DOMONIC development, technical supervision and design insight and suggestions have been supplied by Evminious Damon at NASA Goddard Space Flight Center.

DOMONIC is written in COBOL which makes it hardware independent. The initial version was developed on IBM 360/370 systems and can be adapted to run on CDC and Univac systems. When operating system modifications are made, an excessive amount of program maintenance is required during later stages of operation. Since each manufacturer has developed their own operating system, DOMONIC has been written to be operating system independent. DOMONIC can be used to develop software for any machine written in any language. Software for minis, micros, or large computers written in high-level languages or assembler languages can be developed. DOMONIC can be operated in either a batch mode or an interactive mode. Both versions have been designed to be simple to use by all people involved in the development process. Every effort has been made to minimize restrictions

on programmers who use the system. While DOMONIC is very useful in all phases of software development, the existing software systems can be retrofitted into the DOMONIC data bases allowing the systems to be tested, documented and maintained using DOMONIC.

To better understand how DOMONIC works, let us look at the block diagram of the development process (see Figure 1). Software and development is broken down into development phase and maintenance phase. During development, systems must be specified, designed, programmed, and validated. Data, during all these phases, is deposited into a DOMONIC data base. Every action of all participants in the develelopment process can be monitored through the DOMONIC monitor. Documents for the user, operator, maintainer, and program librarian are automatically validated by DOMONIC. Once data is inserted into the data base by anyone during the development process. this data can be used by others later in the process. For example, the system specifier may input such information as titles, abstracts, systems specifications, testing criteria, block diagrams, etc. The system designer can benefit from any information created by the systems specifier. The designer may add additional information such as system flowcharts, system description, data description, program standards, program design and test procedures, development schedules, etc. The programmer also benefits from the data



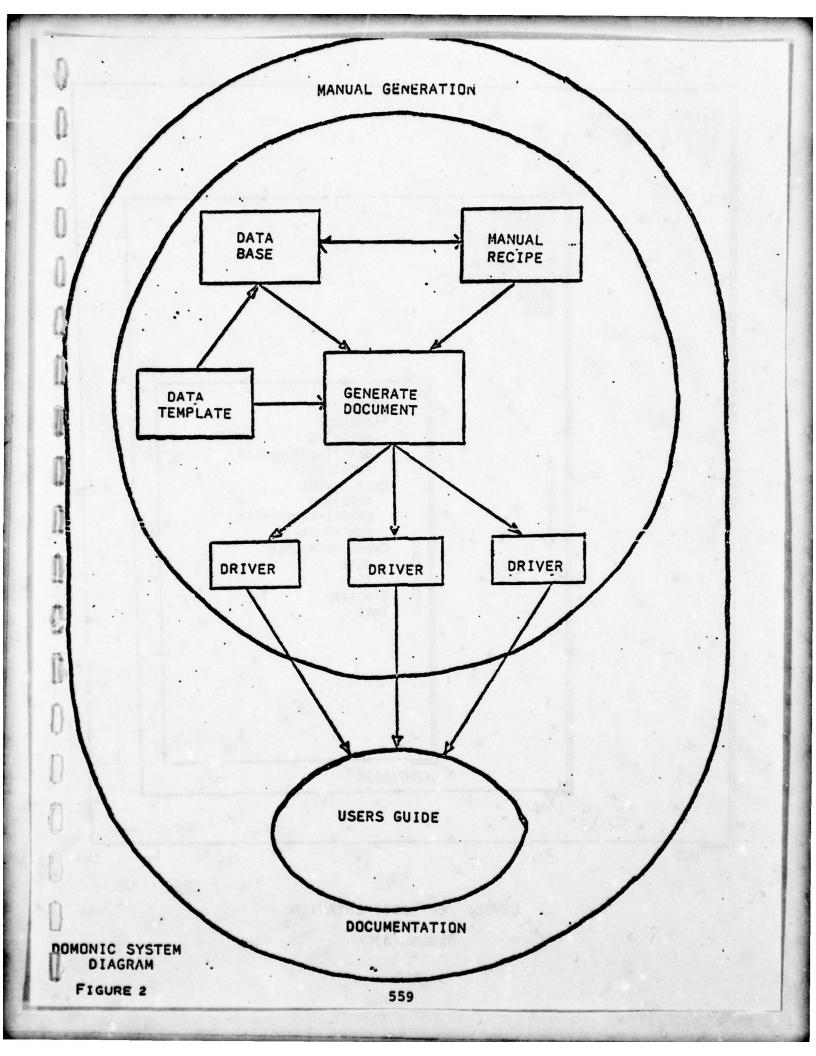
stores as a documentation unit within DOMONIC. The programmer, after entering source code into the system, can automatically generate program flowcharts, cross references, overlay structures, glossaries, etc. The desirable extension to DOMONIC would be to expand the system to automatically produce data layouts. The validator also benefits from previously stored information. Off-the-shelf software packages to validate and test software systems can be called in during the validation stage. Error and test data gathered during the development process and during the validation stage can be used to drive software reliability models.

DOMONIC can be used to assist the documentation process.

DOMONIC can be used as a central depository of all information created for a software system. Information is only entered once which results in the elimination of redundant information storage. For information stored in the data base, the TIDY program flowcharters, text editors, and other documentation aids can be directly driven by DOMONIC. Complexities of job control language unique to individual documentation aids will be transparent to the user. New documentation aids can be easily added to DOMONIC. The major advantage of DOMONIC is that a manager can quantitatively specify and control the programming practices and standards that he would like to be followed during software development and maintenance.

The conceptual block diagram of the DOMONIC system is shown in Figure 2. When a project is initiated, through the concept of data base template, a manager can express all data that is to be collected and stored in the data base. Through the template concept and by using a sophisticated security system, the manager can classify the procedures and standards that should be followed in software development and can later monitor and control the development. The word "manual" in this figure refers to the manual produced automatically as a result of the documentation process. The items that go into a manual are specified through a recipe. The recipe indicates what information will be taken from the data base and used to drive documentation aids to produce a manual for documentation. The same concept can be used to drive software testing and validation packages.

Documentation that can be produced by such a system as DOMONIC can be broken down into system, global, and local documentation (see Figure 3). A great deal of off-the-shelf packages exist to produce local documentation. Local documentation consists of listings, flowcharts, data layouts, cross references, glossaries and text. Similar types of information should also be generated for the global and system levels. DOMONIC can drive such aids as text editors and flowcharters. Special aids have been developed to show subroutine connectivity, cross reference information, overlay maps, subroutine



System Flowchart Block Diagram Text Flowchart global overlay Data Layout Cross Reference Glossary Text Listing Flowchart detailed detail-suppressed functional GLOBAL SYSTEM Data Layout detailed detail-suppressed functional LOCAL Cross Reference data fl ow Glossary Text COMP/ASSM LOAD JOB

LEVELS OF DOCUMENTATION

FIGURE 3

parameters, summaries, etc. Other aids can automatically tidy programs. The system can drive compilers, linkage editors, loaders and utilities which produce information used for documentation. A text editor that could be used on an IBM system is Text 360. DOMONIC can drive other editors when used on Univac and CDC systems. Sophisticated subroutine connectivity maps can be drawn which are similar to the hierarchial diagrams used as a part of HIPO charts. Special documentation aids to produce exhaustive cross reference information for FORTRAN, COBOL, PL/1, and assembler language programs have been produced. A TIDY program can be used to automatically renumber statements, insert spaces for clearer reading, and indent sections of segments relating to the same control sections.

Overlay maps and a detailed summary of all parameters can be automatically generated. Compilers, linkage editors, loaders and utilities generate a considerable amount of information that can be used to document a program.

Although there are a number of automatic flowchart systems available, these systems cost a significant amount of money each time they are installed on a new computer. An advanced flowchart system has been developed for use in DOMONIC. The system can draw any size flowchart symbol. The symbol can be any shape and can be compactly placed on any size page.

The system has been designed to be completely language independent, but the system has been initially implemented to draw flowcharts of FORTRAN programs. The initial version of the system can draw detailed flowcharts or flowcharts with details suppressed. The system has been implemented using COBOL.

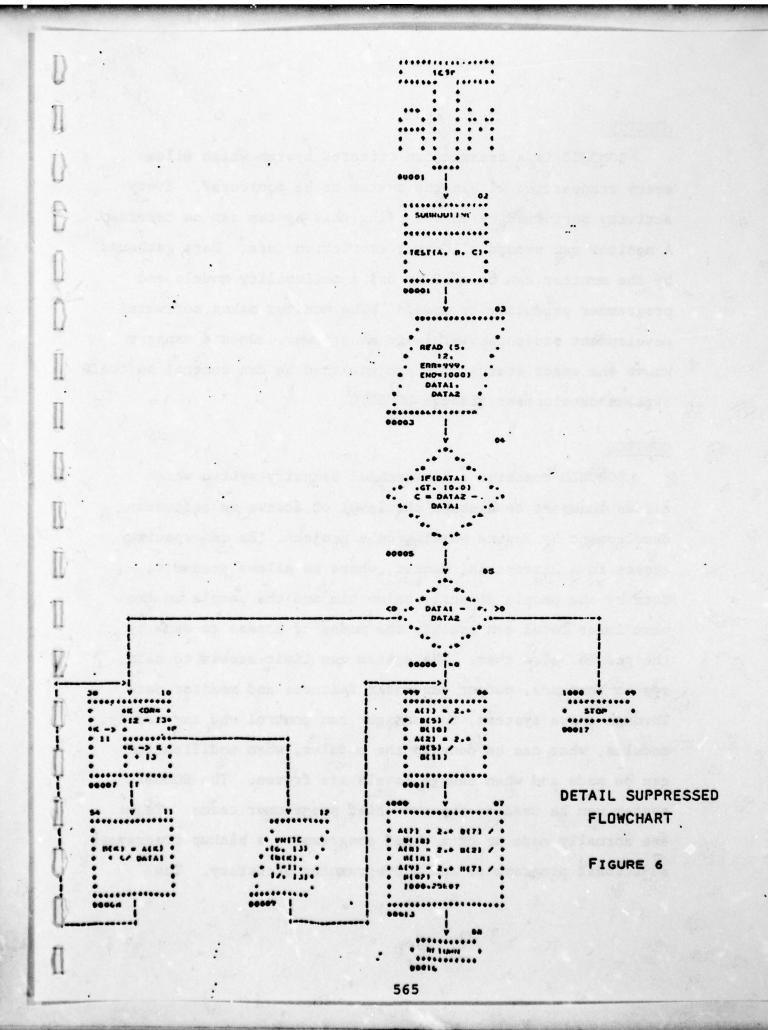
A flowchart has been drawn of the simple FORTRAN program shown in Figure 4. Notice statement three has a READ statement that can transfer to three different places. Statement five is a logical IF statement and statement seven contains a DO loop. A detailed flowchart of the program in Figure 4 is shown in Figure 5. The entry symbol that says TAM was created to show the versatility of symbol shape. Normally, the symbol would be an entry symbol.

Notice that the logical IF statement is created by using two symbols. The DO loop has been created by using four symbols. Figure 6 shows a details suppressed flowchart. The READ statement has been compressed down to a single input/single output symbol and the logical IF is contained in a single diamond. The DO statement has been compressed to a single symbol.

```
00001
              SUBROUTINE TEST(A.B.C)
              DIMENSION A(20), B(20)
00002
               READ (5,12.FRR=999, END=1000)DATA1, DATA2
00003
           12 FORMAT (2F5.0)
00004
               IF(DATA1.GT.10.0)C=DATA2 - DATA1
00005
               IF(DATA1-DATA2)30,999,1000
00006
           30 00 54 K=11,12,13
00007
           54 E(K) = A(K) + C/DATA1
80000
              WRITE(6,13)(8(K). I=11.12.13)
00009
           13 FORMAT(10F10-0)
00010
          999 A(1) = 2. + 6(5) / 6(10)
00011
               A(2) = 2. * B(5) / B(11)
00012
         4000 A(7) = 2. # B(7) / B(18)
00013
               A(8) = 2. + B(2) / B(18)
00014
               A(9) = 2. # B(7) / B(02)
                                             + 1000.25E07
00015
00016
              RETURN
         1000 STOP
00017
00018
              END
```

A FLOWCHART OF A FORTRAN PROGRAM
FIGURE 4

\*\*\*\*\*\*\*\*\*\*\*\*



## MONITOR

DOMONIC is a transaction oriented system which allows every transaction within the system to be monitored. Every activity performed by anyone using this system can be recorded. A monitor can record all error correction data. Data gathered by the monitor can be used to drive reliability models and programmer productivity models. The monitor makes software development projects visible to management. Once a manager knows the exact status of a project then he can control software systems development through DOMONIC.

#### CONTROL

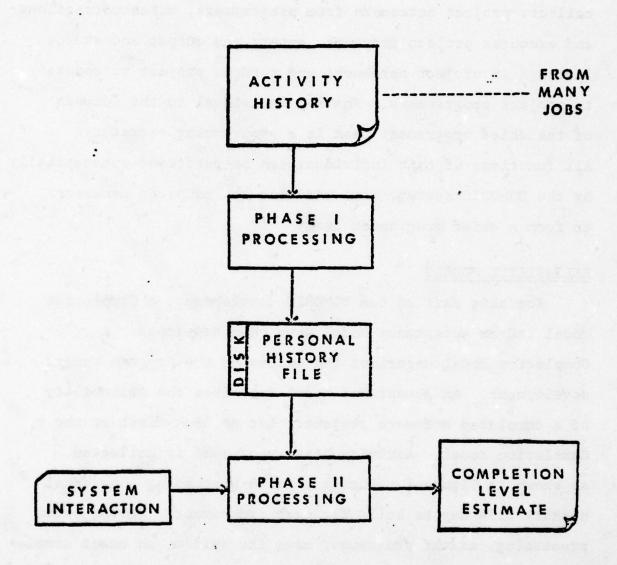
DOMONIC contains a hierarchial security system which allows managers to dictate the level of access to software development by anyone working on a project. He can specify access in a hierarchial manner, where he allows access to data by the people directly below him and the people on the next lower level can specify the modes of access to data by the people below them. The system can limit access to data, system commands, output generator features and monitor data. Through these systems, the manager can control who can modify modules, what can be done to the modules, when modification can be made and when change levels are frozen. The DOMONIC system can be used to improve chief programmer teams. Teams are normally made up of a chief programmer, a backup programmer, additional programmers and a programming secretary. The

program secretary maintains the program production library, collects project notebooks from programmers, makes corrections and executes project programs, enters new output and source listings in project notebooks and returns project notebooks to project programmers. The key individual to the success of the chief programmer team is a programming secretary.

All functions of this individual can be performed automatically by the DOMONIC system, thus reducing the manpower necessary to form a chief programmer team.

## RELIABILITY MODELS

For this part of the DOMONIC development, a Completion Model and an Acceptance Model have been developed. A Completion Model describes the status of the program under development. An Acceptance Model describes the reliability of a completed software project. Let us look first at the Completion Model. Activity history of jobs is collected, as shown in Figure 7. During initial processing a personal history file can be built for each programmer. During later processing, as the programmer uses the system, an exact completion level estimate is used for a system. The completion model takes advantage of the fact that different activities are normally performed during different stages of development (see Figure 8). For example, during early development, source code is frequently added, some of the source code is modified, no execution is performed and there is no need to remember programs and produce other cosmetic edits. This



I

BLOCK DIAGRAM OF COMPLETION STATUS MODEL
FIGURE 7

# Activities

Add Source Code	Modify Source Code	Executions	Renumbering and Cosmetic Edits
Often	Some	None	None
Some	Some	Some	Some
Seldom	Seldom	Many	Some

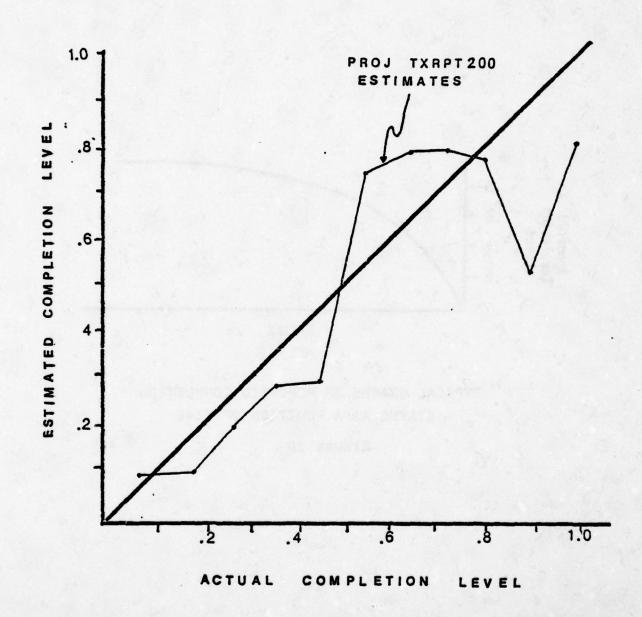
Early Development
Mid-development
Final Testing and
Documentation

PROGRAMMER ACTIVITIES DURING DIFFERENT DEVELOPMENT PHASES

FIGURE 8

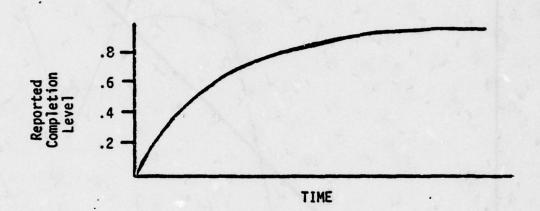
profile is different for the mid-development phase and the final testing and documentation phases. For example, in the final phase, source code is seldom added and the source code that is there is seldom modified. Programs are frequently executed and documentation aids to renumber statements and form other cosmetic edit are used. The personal history for each activity of a programmer is created. From this, profile completion can be estimated for each module. For example, completion of a single module is demonstrated in Figure 9 which plots estimated completion level versus actual completion level. A composite of the completion level for all of the modules can be plotted versus time to estimate the completion of a total project (see Figure 10).

In contrast to the Completion Model, which can estimate how close to completion the system is, the Acceptance Model can be used to determine the reliability of a finished product. This model uses conventional reliability measurement technology. In development of the acceptance model, emphasis has been placed on making the technology understandable. Its main use is in the validation of software products. A basic approach is to first decide upon the desired reliability levels. Next, the number of tests required must be determined. Actual tests must then be conducted and the number of successful tests must be recorded. After the tests are completed, the decision to accept or reject a product must be made. An example of



ESTIMATED VS. ACTUAL COMPLETION LEVELS

FIGURE 9



TYPICAL GRAPHS OF REPORTED COMPLETION
STATUS AS A FUNCTION OF TIME
FIGURE IO

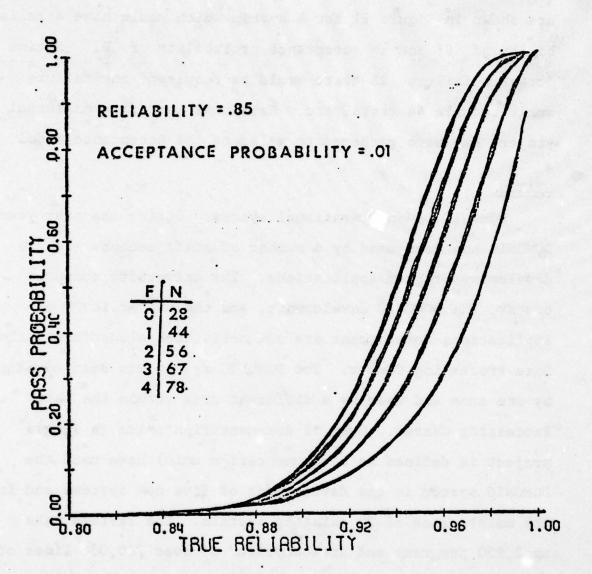
the curves which plot pass probability versus true reliability are shown in Figure 11 for a system which would have a reliability of .85 and an acceptance probability of .01. Notice for zero failure, 28 tests would be required, one failure would require 44 tests, etc. Extensive test data on actual project has been gathered to validate the Acceptance Model.

## STATUS

DOMONIC is an operational system. During the past year, DOMONIC has been used by a number of staff members who are developing program applications. The university computer center, the DOMONIC development, and the university applications development are all activities placed under the Data Processing Center. The DOMONIC system has been developed by one area and used by a different area within the Data Processing Center. Over 35 documentation units (a single project is defined as a documentation unit) have used the DOMONIC system in the development of five new systems and in the maintenance of 30 existing systems. The systems make up 2,950 programs and are comprised of over 700,000 lines of code.

# NEXT PHASE

As mentioned earlier, the DOMONIC system was developed to operate in both batch and interactive modes. While the interactive version operates effectively on the nonsaturated



.

-

PASS PROBABILITY - TRUE RELIABILITY

FIGURE II

Amdahl 470 V/6 at Texas A&M, the saturated IBM system at Goddard will have difficulty responding in the interactive mode. During the next phase, critical program modules will be redesigned to speed up the interactive response time. Another approach to proving response time will be to distribute the critical interactive operations to inexpensive microcomputers that will interface to the operating system using standard protocols and drive up to 16 terminals. The improved version to the system that is being delivered to operate on the IBM system at Goddard will be adapted to operate on CDC and Univac computers at other locations. During this phase, additional effort will also be expended to improve and refine the software reliability models.

There are a number of projects at the Data Processing Center related to DOMONIC development. These projects are:

- A COBOL statistic collector is being developed for Rome Air Development Center which can do static analysis of COBOL programs and be expanded to do dynamic analysis.
- The building and testing of a system to randomly generate test data to validate programs has been completed.
- A comprehensive analysis of program connectivity to determine optimal overlay system and virtual memory working sets has been completed.

- Comprehensive study of program complexity has been completed which includes a rough quantitative measure of program complexity.
- 5. Design specification languages are being investigated.

1

6. Line routing and placement routine for general drawing systems are being examined.

## DESIRABLE ENHANCEMENTS

The generalized flowchart system can be expanded to automatically produce HIPO charts. Extensive data should be gathered to investigate programmer productivity. A project evaluator should be developed to automatically determine the status of a software system delivered to a contracting agency. The evaluator system could predict software reliability using a model such as the Acceptance Model. Packages that automatically test software could be driven by the system. The delivered system could automatically analyze a program to determine whether structured programming techniques were used. Also, the system could be automatically checked to determine whether appropriate programming practices and standards were followed.

There has long been a desire in both hardware and software development to produce a high level design specification language. Such a language could be developed for the DOMONIC system.

## ACKNOWLEDGEMENTS

Numerous staff members and graduate students have participated in the design and development of the DOMONIC system. Evminious Damon at NASA Goddard Space Flight Center has given major encouragement and advice throughout the DOMONIC development. Louis DeVito and Mike Quick deserve special recognition for their contribution to the DOMONIC design and development. Susan Arseven supervised the initial design and implementation of the first version of the system. Pete Marchbanks has supervised the latest design and development of the current version of the DOMONIC system. Jean Zolnowski and Jimmy Moore designed and developed the flowcharting system included in the DOMONIC system. Roger Elliott has been involved in most phases of the system design and has supervised the development of the reliability model. Larry Ringer has acted as consultant on the development of the reliability models. Numerous other graduate students, faculty members, and computer center staff have also made significant contributions to this development.

DBS/1f/mbt 08/17/77

## DESIGNING A SOFTWARE MEASUREMENT EXPERIMENT\*

Victor R. Basili and Marvin V. Zelkowitz
Department of Computer Science
University of Maryland
College Park, Maryland 20742

## Overview of the Laboratory

The Software Engineering Laboratory has been established at NASA Goddard Space Flight Center in cooperation with the University of Maryland in order to study the effects of various software development methodologies on software projects in a real environment. Projects are being studied at three levels of control and detail: screening experiments, semi-control experiments, and control experiments.

The screening experiments involve the collection of data on the development of several ground support systems used to control spacecraft operations. Their purpose is to determine how software is being developed now, supply data for a data bank of information to classify projects for future reference and public availability, expose the differences between the theoretical application of a methodology and its practical implementation, discover what parameters can be validly isolated, expose the parameters that appear to be causing major problems, and discover the appropriate milestones and techniques that show success under certain conditions.

The semi-controlled experiments involve the collection of data on several similar attitude determination systems using prespecified development methodologies. Their purpose is to extend the screening experiments by permitting tighter control and to permit the monitoring of different development methodologies, comparing the various factors affected by the different methodologies.

Research supported by grant NSG-5123 from NASA/Goddard Space Flight Center to the University of Maryland.

The controlled experiments involve the development of duplicate programs analysis and data base systems with fairly rigid control of many of the factors affecting software development. The purpose here is to design usable experiments for various development factor evaluation that can yield statistically valid results.

The activities of the Laboratory include data collection, data processing, data evaluation, and courses on methodology. Data collection activities include forms filled out at various points in the development, covering various aspects of the product; interviews used to validate the forms and collect information not easily filled out on a form; and automatic collection of data via the existing system and program development library when available. The data processing activity involves entering the data into a computerized data base and screening for accuracy. The data evaluation activity involves the analysis of the data collected with emphasis on management, reliability, and complexity. This analysis deals with simple raw statistics, in the forms of counts, sums and averages, derived statistics in the form of correlations and multivariate analysis of the relationships between various factors, and the development of measures based on theoretical models of program behavior and complexity. The measures will be computed from combinations of the raw statistics and derived statistics.

The simple raw statistics include project attributes such as type of processing, instruction size, code size, methodologies and techniques used, resources, total man hours, calendar time, types of computer access, automated aids and tools, usable items from previous projects, milestones, etc. Also included are counts of total number of program changes, total errors due to various factors, such as misinterpretation of the requirements or specification, total time spent in each phase of development, errors found using various techniques, total computer usage, etc.

COMPUTER SCIENCES CORP ARLINGTON VA

SOFTWARE PHENOMENOLOGY - WORKING PAPERS OF THE SOFTWARE LIFE CY-ETC(U)

AUG 77 B ELKINS, L HUNT

DAHC26-76-D-1006 AD-A053 014 UNCLASSIFIED NL 7 OF 8 ADA 8 053 014 511

Derived statistics are used to find relationships among various factors.

These would include the relationship between the number of errors and program size, between errors plotted against time in development, between various estimates (e.g., cost, time) which have been reevaluated at various points in the development, and actuals at the end of the project, noting points at which the estimates become realistic, and between errors distributed according to the amount of time they remained in the system, etc.

At the other end of the spectrum, measures are being developed that are associated with different models of program behavior and complexity which are being tested against the statistics calculated above. For example, various measures derived from models of complexity will be compared with development time, errors and subclassifications of errors, subjective views of the complexity of various modules, etc., to determine if these measures and models can be validated in a real environment. Measure of complexity include the number and significance of control paths and data bindings based on various criteria for hierarchical decomposition of the program into elementary subschemes using different formulas to combine their individual evaluations (SULL, 73), (LING, 77).

This paper deals predominantly with the purpose, design and problems involved in organizing a controlled experiment. The purpose of the controlled experiment is to isolate the effect of specific factors, in our case mostly software development techniques, on the development process. The idea is to design as airtight as possible a valid, usable experimental paradigm for methodology evaluation. Clearly, regardless of the statistical significance of the results of one experiment, one needs to run several such experiments to accumulate substantial confidence in the interpolations made from those results.

## Design of the Control Experiment

The current design involves the use of two programming teams, both assigned the same project tasks. Both groups, the experimental (1) and control (2) will be treated in identical fashion as much as possible, with the sole systematic exception being the experimental treatment. The experimental group will be assigned a task A which will be used as an indicator of their "normal" behavior. They will then be trained in a specific methodology. The methodology will be reinforced in a session analyzing the development task A but in the new methodology. They will then be given a second task B in which to practice the methodology. They will then be assigned task C with the specification that they use the newly learned and practiced techniques. In parallel, a control group, group 2, will also be given task A, followed by session analyzing the development of A using the methodology performed on A, followed by task B, followed by task C. The difference is that group 2 has no training session.

This design gives us several points of comparison. We can discover any differences in the capability of personnel by comparing data from project A for both groups. The two groups can then be more honestly compared on project C by statistically controlling for any differences observed in project A. The design was developed to minimize several problems which often jeopardize the validity of experimental designs. (CAMP, 66).

There are several problems which affect the internal validity of the experiment; i.e., extraneous variables which can produce effects confounded with the effect of the experimental manipulation. There is also the problem of external validity; i.e., to the extent that the experiment is valid, is it possible to generalize the conclusion drawn from the experiment. We will discuss several of these problems here and defend the design of our control experiment with respect to these factors later.

External validity is affected by: the reactive and interactive effect of testing, in which a pretest might increase or decrease the respondent's sensitivity or responsiveness to the experimental variable and thus make the results obtained for a pretested population unrepresentative of the effects of the experimental variable on unpretested respondents; the interaction effects of selection biases and the experimental variable; reactive effects of experimental arrangements, which would preculde generalization about the effect of the experimental variable upon persons being exposed to it in nonexperimental settings; multiple-treatment interference, which may occur whenever multiple treatments are applied to the same respondents, because the effects of prior treatments are not always erasable or ignorable.

### Implementation of the Design

This section deals with the specific organization and implementation of the control experiment design in the NASA Goddard environment. The tasks and training will be discussed first and then techniques for handling specific "real world" problems will be discussed.

The experimental group and control group will each consist of three programmers, working approximately half time on each of the tasks assigned. This is the normal operating environment at Goddard; most programmers work on more than one task at a time. The groups began task A approximately one month apart. Task A consists of two projects; a FORTRAN static analyzer and a human resource scheduler. The code analyzer will generate several statistics about FORTRAN source programs and is estimated to be a two man month effort. The human resource scheduler is an interactive data base system to enter, update and report the scheduling of people on tasks. It is estimated to be a three man month effort.

After both subtasks of task A have been completed, the experimental group will undergo training in a particular set of structured design and programming techniques. The training will consist predominantly of a one week course which will cover the topics shown in the following outline:

- Day 1: Introduction, functional model of structured programming,

  process design language, reading structured code, laboratory
  exercises in the given techniques.
- Day 2: Management and estimating problems, proving correctness of structured design, writing structured programs, laboratory exercises.
- Day 3: Documentation, stepwise refinement, stepwise reorganization, laboratory exercises.
- Day 4: Organizing for structured programming, modularization, top down development, stubs, program development libraries, walk throughs, laboratory exercises.

Day 5: Project control, iterative enhancement, case study analysis, laboratory exercises.

This course is offered on a continuing basis and the experimental group will be members of a larger class of approximately 25 people. This approach was chosen to help minimize any special group interaction effect on the experimental group.

In order to help insure that the group understands the new methodology, there will be a one or two day special training session with the group in which the design of the FORTRAN code analyzer from task A will be reanalyzed within the framework of the newly learned methodology. To compensate for this special group activity and design analysis instruction the control group will be exposed to a similar experience. Even though they will not be given a course, they will have a similar one or two day session in which they will review their design of the FORTRAN code analyzer with respect to their particular design methodology.

Clearly, it takes time and practice to achieve a reasonable amount of skill in the use of a new set of techniques. It is difficult for us to estimate how long it will take for the programmers in the experimental group to acquire a sufficient amount of skill to show an improvement, if there is going to be one, over the previously acquired technique skills, before we can attempt to test the use of this new skill. To compensate at least partly for this uncertainty, task B will be assigned. Its main purpose is to give the experimental group experience with the newly learned methodology. Task B is a NAMELIST processor, a prog. A that will recognize S/360 FORTRAN NAMELIST statements and convert them into equivalent non-NAMELIST type FORTRAN I/O statements. It is assumed that this task is a one man month of effort. Both groups will perform task B.

Task C will be used to perform the comparison between the two groups. It also consists of two subtasks: a contractor financial report program and a control monitor. The contractor financial report program is a data base analysis system similar to the human resource scheduling program of task A. The control program is a real time, interactive attitude support subsystem of six man months effort. This provides one project similar to one developed without the new methodology as well as a totally new type of project.

A major advantage of this design is that it permits us to analyze the differences between the two groups with regard to task A, working in their own self-defined environment. Measures on task C can then be statistically adjusted by performance on task A.

During the entire duration of the control experiment, regular meetings will be held with each of the groups to answer any of their questions on how the reporting forms should be filled out and to gather extra information that is either not clearly specified on the form or was not obtainable from the forms at all. These interviews will also be used to help validate the information that is on the filled out forms. To help eliminate any biasing here, all interview questions will be written down and the same questions will be asked of both groups.

There are several problems that arise in trying to implement the design in a "real world" environment. A couple of these will be discussed. One problem that arises in the development of real large scale projects is that the specifications are not always well defined. In this case a project monitor, who is not a member of the group, will be used to answer questions about the specifications and make decisions about what is really meant. There will be two different project monitors, one for each group. This is to help guarantee that any information learned by the monitor will not be passed on unconsciously to the other group. When a question arises about interpretation, raised by one

of the groups, the contract monitor will consult with the other contract monitor to see if it has been resolved, if not he will resolve it. If it has been resolved, he will be told how it was resolved and then give a resolution to his group. Thus, if the methodology used by one group helps them understand where the specifications are unclear, earlier in the development, the other group will not benefit from this information. The only problem here is that the two groups may develop slightly different systems. In this case we will go to outside arbitrators and to the project monitors to determine that both systems meet user requirements satisfactorily.

There is also the problem of keeping the experiment secret. That is, we do not want either group to know that the other is doing the exact same task and that they are being compared on certain measures of management control, reliability and the complexity of the final project. For this reason, we selected programmers from isolated, but hopefully similar groups. One group consists of programmers employed by a contractor. The other group is internal to NASA. They have both been told that they are being carefully monitored because we are interested in the effect of the forms and whether they can be filled out accurately. They know another group is also doing a "similar" set of tasks and our interest is in whether the forms get filled out in the same way.

#### Defense of the Design

In this section we will try to defend the design and implementation of the experiment from the points raised by Campbell and Stanley that threaten the internal and external validity of the experiment.

The effect of history has been minimized by organizing both developments in as close a time period as possible. Actually, as we stated above, one group did start about a month ahead of the other but we do not feel this will make much difference in a project that has an elapsed calendar time of 9 to 12 months.

The course of events over time is identical for both groups, except for the experimental group's training session.

Maturation was seen to have the potential of being a real problem in this kind of research since the more one knows about a particular project and the more a group works together, etc., the better they may perform. For this reason it would not be proper to compare the performance of a single experimental group on tasks A and C, even if they were similar projects. The proposed design, using a control group, has the effect that the maturation factor and learning curve affects both groups similarly. The experience of the groups is kept similar, especially with the design analysis sessions and the development of task B.

With regard to the testing factor, there is no explicit testing that would influence either group, except possibly for the experience of filling out of the forms and the interviews. However, in those cases both groups have the same experiences.

The variability of instrumentation is kept to a minimum by using the same forms for both groups and similar interview questions.

To minimize the possible effect of <u>statistical regression</u>, we tried to choose average programmers for both groups. That is, we did not choose super programmers or poor ones. This rating was decided based on general knowledge of the programmers' performance on previous tasks.

Because of the need for some secrecy and problems with contracting for specific people, the <u>selection</u> variable presented more of a problem than the others. We could not make a random selection of programmers. What we did do was try to match an internal group to the contractor group. In any case, simple effects of selection biases may be letermined by examining our measures with respect to project A and any sum effects controlled in later analyses.

To minimize problems with experimental mortality, we chose projects of minimal size, the whole development lasting somewhere on the order of nine calendar months. We tried to make sure the participants would be available for the full duration of the project, barring any unforeseen disasters. Clearly, there is no way to totally control this variable.

Based on the fact that the two groups are similar in terms of background and capability, it is hoped that the <u>selection-maturation interaction</u> will not be a problem; that it, it should affect both groups the same in that there should be similar maturation for both groups as they are exposed to the sequence of tasks.

Clearly, the <u>reaction or interactive effect of testing</u> is a factor here that will affect generalizing the results. It is not normal for projects to be studied to this degree of detail. There will be an effect related to the filling out of the forms and the knowledge of the participants that they are being studied, even though they do not know the evaluation technique, that will not be able to be factored out. For this purpose, we plan to collect data on several projects that have already been completed but were not monitored so closely. In this way, it will be possible to get general data to compare with the data collected on the control experiments and see if they are performing within the general set of standards.

With regard to interaction effects of selection biases and the experimental variable, both groups appear to be rather typical of the general population of programmers and there appears to be no special interest on the part of either group in the new methodology.

The decision to analyze the development methodologies was not based on the belief that there was a specific methodology that was best but on the fact that analysis should generate a better understanding and therefore help in creating an improvement in any methodology. Thus, there was no conscious predisposition

toward a particular methodology on the part of the experimentors that would give any advantage or disadvantage to a particular methodology being tested. In fact, the specific structured programming methodology being tested is only one of many methodologies and variations that are to be tested. It is also true that the programmers undergoing the experiment do not know what the experimentors are looking for and therefore cannot behave in any particular way to affect the results of the measures. As stated earlier, there will be some reactive effects of experimental arrangements based on the programmers knowledge that their environment, due to the forms, is different than the normal environment. However, we do have an advantage over most experiments of this type in that we are trying to study the software development process in its "natural" environment rather than in a classroom, student programmer situation.

Lastly, there should be no <u>multiple-treatment interference</u> since our studies should involve fresh population samples of programmers. None of them have been used in earlier experiments. Hopefully the programming community is large enough that there will always be a fresh supply of "uncontaminated" subjects.

#### Conclusion

From what has been said, it is clear that it is difficult to design a controlled experiment in a real environment. There are too many factors that cannot be measured accurately or completely controlled, such as programmer ability. Designs must be devised that minimize these problems. It is difficult to implement a design since it is difficult to control all the factors in a real environment; e.g., contracting software so that it be-performed in a certain way, guaranteeing that the same people will be available throughout the experiment.

However, the authors believe that it is important to perform such studies, creating the most controlled environment possible, if we are ever to gain any real insights into the effects of various software methodologies on the

Parent of the last

development process and the resulting product. We feel the design proposed in this paper is a reasonable first attempt at approximating an effective experimental design in the given environment. It is clear that much will be learned about the development of program development experiments and it is hoped that a great deal will be learned about the development process. We hope to continue to report on the progress, problems, errors and successes of running controlled experiments in this area and hope that we can gain from the efforts of others as well.

#### Acknowledgements

The authors would like to thank Robert Reiter at the University of Maryland for his work on the basic controlled experiment design currently in use.

### References

- (BASI, 77) Basili, Victor R., Zelkowitz, Marvin V., et al., The Software Engineering Laboratory, University of Maryland Computer Science Technical Report, TR-535, May, 1977, 104 pages.
- (CAMP, 66) Campbell, D. T., Stanley, J. C., Experimental and Quasi-Experimental Designs for Research, Chicago, Rand McNally Publishing Co., 1966.
- (LING, 77) Linger, R. C., Mills, H. D., Structured Programming Theory and Practice, Addison Wesley, 1977 (to be published).
- (SULL, 73) Sullivan, J. E., Measuring the Complexity of Computer Software, MITRE Corp. Report MTR-2648, Vol. V, June 1973.
- (ZELK, 77) Zelkowitz, M. V., Basili, V. R., Operational Aspects of a Software Measurement Facility, Proceedings of the Software Life Cycle Management Life-Cycle Workshop, August 1977.

## Operational Aspects of a Software Measurement Facility\*

Marvin V. Zelkowitz and Victor R. Basili Department of Computer Science University of Maryland College Park, Maryland 20742

The University of Maryland, in conjunction with MASA Goddard Space Flight Center has organized the Software Engineering Laboratory for the purpose of Measuring and evaluating techniques used in the development of spacecraft related software (Basili-Zelkowitz). After being in operation for a year, we are now starting to process the data necessary to perform these evaluations from about a dozen MASA projects. From the experience gained by collecting data from these projects, our data gathering process has evolved into a more effective operation.

As MASA software is developed auxilliary data for the laboratory are produced which passes through four distinct phases: project development, data collection, data processing and data evaluation. This report describes the organization of the laboratory and how we have handled some of these operational aspects.

## DATA FLOW

MASA/GEFC launches approximately 25 unmanned satellites per year and most of these include the development of ground support software requiring about six man years of effort. These software projects generally take about a year and involve from 8 to 15 people, most of whom are working on several such projects simultaneously. The projects are supervised by MASA/GEFC personnel with most of the actual development by employees of an outside contractor. In an era of tight budgets, the purpose of the laboratory is to investigate these projects and find techniques that will lead to increased productivity

<sup>\*-</sup> Research supported in part by grant MSG-5123 from the Mational Aeronautics and Space Administration - Goddard Space Flight Center to the University of Maryland.

on a constant yearly budget.

BASA and contractor management were eager to participate in this study, but were concerned that the impact on current development schedules and costs be minimal. In order to minimize this impact, we decided early to use a set of reporting forms as the besic data gathering mechanism. It was hoped that the forms could be filled out easily, would not interfere with current techniques, and would not involve much overhead, yet would still give an accurate picture of development progress.

The laboratory was organized in August, 1976 via a research grant to the University of Maryland. Currently, personnel include 5 employees of MASA/GSFC, 2 faculty and 9 students at the University and approximately 75 programmers employed by the contractor on the projects that we are monitoring. The two major tasks of the Laboratory are project management and data evaluation. The day-to-day operational aspects currently are broken down into four areas:

- 1. MASA/CEFC personnel are the interface between the contractor and the laboratory. They are responsible for project development and for contractor compliance with laboratory requirements. They are supervising spacecraft projects as part of their normal workload in their normal environment.
- 2. The main research responsibilities are being performed by the authors of this report along with several graduate students. In addition some of the EASA personnel are involved in this activity.
- 3. In order to evaluate the collected data, laboratory software for a PDF11-based system is being developed. This is being performed by two student programmers under faculty supervision.
- 4. The day-to-day data collection is currently being handled by one MASA employee and four university undergraduate students under the supervision of a graduate student.

Given this structure, data flows through the laboratory as outlined by figure 1:

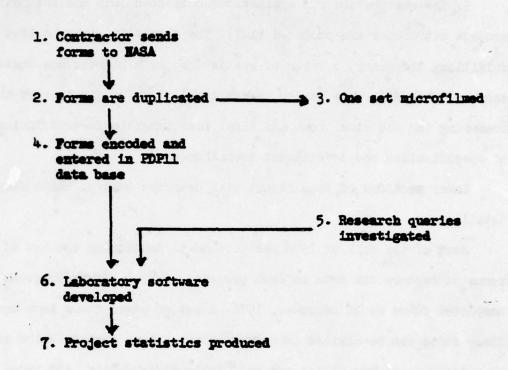


Figure 1. Laboratory Data Flow

- 1. The contractor, about once a week, sends all forms for all projects that have been filled out to MASA/GEFC (Figure 1.1).
- 2. The forms are implicated at NASA (1.2), and one set is sent to be Terro-filmed for archiving (1.3).
- 3. The other set is sent to the University of Maryland where it is encoded onto coding sheets and entered into a PDF11 data base (1.4). As will be diplained later, the data is read four times before being entered into the data base. Thus we believe that we have eliminated virtually all clerical errors in the process.

While the above are continuing operations, the following tasks are of a research nature and occur periodically:

4. The graduate students develop queries about aspects of the data to be tested against our collected data (1.5). At the same time, special purpose laboratory software is developed for use in the laboratory (1.6). This includes enhancing the data base system, programs to validate the data and special purpose routines, such as a plotting package.

5. The queries are run against our collected data and statistics about project attributes are produced (1.7). The output is either a list of factors fulfilling the query, a table of statistics, or a plot of one factor versus another (generally some factor versus time). Some of the factors that we are measuring include size, cost and time. Less tangible factors include level of specification and development technique used.

Later sections of this report will describe most of these steps in greater detail.

Most of the fall of 1976 was devoted to developing the set of reporting forms to capture the data on each project, with the first projects submitting completed forms as of December, 1976. A set of seven forms have been developed. These forms can be divided into two classes—those that describe attributes of a project and are filled out relatively infrequently, and those that monitor progress and are filled out frequently. The forms that describe attributes of a project are the following:

- 1. General Project Summary. This form is filled out at each project milestone, and there are from five to ten milestones in the lifetime of a project (every six to eight weeks). This form describes the overall project structure, and the techniques used in its development. Such factors as project size, complexity, estimated milestones, required standards and documentation are given. How these change over time is one of the areas that we are investigating.
- 2. Component Summary. This form is similar to the general project summary except that it describes only one small section of a system (e.g., function, subroutine, COMMON block, etc.). This form is filled out when that particular component is first designed in the design phase of the project and again when it is completed.
- 3. Programmer Analyst Survey. This form is used to collect general background information on project personnel, and is filled out only once by each programmer. It is used to compute a profile of project personnel in order to

be able to compare two projects more equitably.

The forms that monitor project progress are the following four:

- 4. Resource Summary. This form is filled out weekly by the project manager, and it lists the hours spent by each individual on the project. Since this information is already collected for accounting purposes, little additional effort was added. In addition, MASA/GSFC adds to this form a count of the total computer time used by the project and the total number of runs, as reported by the MASA computation center.
- on the project. It lists the number of hours spent on each component and the activity performed on that component (e.g., design code, test, etc.). Besides some of the obvious statistics we expect to obtain (e.g., number of hours spent in design vs. test vs. code, etc.) we can also use this form to verify the techniques used in project development. Unfortunately terms like "top down," "structured programming," "walk throughs," and "code reading" have become general purpose buzzwords with different interpretations. One of the results that we expect to get from the component status report is to be able to classify the techniques that are being used rather than the ones we are told are being used. We can obtain this information by noting when certain activities occur for each component.
- 6. Computer Program Run Analysis. This form is filled out for each computer run. It is mostly a checklist of activities performed (e.g., compile, execute, utility, etc.). In addition, the error message and component terminating the run is listed. This will be correlated with the following change report form.
- 7. Change Report Form. For each change in the source code for a component (either to fix an error or to implement some change in specifications or design) a change report form is submitted listing the change and the reason for it.

  Since a form of this type has been in use for some time in this particular MASA environment, little additional work was added to existing MASA projects.

  One limitation, however, is that changes are not reported for any component

until that component is placed into the project library. Thus while we are getting all of the errors found in integration testing, we are not (except for a few of the monitored projects) getting much from the module design and development phases.

#### DATA COLLECTION

Broadly stated the purpose of the data collection phase is to transmit the raw data on each monitored project to the laboratory for processing and analysis. This aspect necessitates the interaction of three different groups of individuals. Needless to say, although in the long run all these groups are interested in better software methodologies, each have differing immediate goals.

The primary requirement for each monitored project is to develop spacecraft support programs; therefore, the primary goal of the MASA/CEFC personnel is the management of project development on schedule. However, actual program development is by an outside contractor whose main goal is to deliver such projects on time in order to meet contractual obligations. Finally the third group involved in the laboratory is the University of Maryland which is primarily interested in the analysis and measurement of the development process.

The use of three different groups has both negative and positive attributes towards our goals. Clearly there are communications problems. Any changes in reporting requirements must first be conveyed to MASA/GSFC who in turn must inform the contractor. Similarly any unformation from the contractor is first routed through MASA.

On the other hand, this situation increases the impartiality of the research group with the contractor group. Since we are independent of MASA, we are not involved in rating contractor performance and evaluating contractual arrangements. We seem to have the contractor's trust which we believe gives us more accurate data to work with. As part of this impartiality, we are careful not

to report back any data that can be identified to specific individuals. All programmer names are encoded in our data base, and all data reported back to the contractor (and to MASA also) is of a summary nature.

# DATA PROCESSING

In the data processing phase, the data on the projects must be transcribed onto coding sheets and entered into a PDF11 data base. Such issues as data validity and completeness are of primary importance in this phase.

The raw data is checked four times before being entered into the data base. The forms are first encoded onto coding sheets and then checked for errors by a second person. The coding sheets are then typed into the computer and run through a special program which further checks for errors (e.g., format errors or improper fields such as a wrong component name in a certain project) and converts the data into a format suitable for inclusion into the data base. The output of this program is again verified against the coding sheets to further check for typing errors. By the time that the data is in our data base we are confident that what has been entered is an accurate translation from the original forms. Whether these forms accurate describe the actual project, however, is another issue which will be discussed shortly.

Forms validity is perhaps the hardest problem that must be tackled. Unfortunately there are few general standards applying to all MASA/GSFC projects, and one of the requirements on the laboratory is to measure current techniques without impacting most projects. Thus forms are filled out with varying degrees of completeness, and we must be careful in applying results across several projects. In one such case, there is about a factor of three in number of forms submitted between one project and others of a similar size.

The impact on projects brings to mind two examples of the kinds of problems we are faced with. One project manager has been reluctant to participate since his project is late and he sees no apparent benefit from participating in this research. While this group blames the forms for any increased lateness, they were already behind schedule before the forms were instituted. MASA/GSFC believes that this group has always been poorly organized and the problems with the forms only points out this problem and is not the cause of it. For well managed projects, the filling out of the forms is no more complex than any other administrative detail of a task.

This example shows one of the dangers in interpreting our collected data. Since forms may be the first to go when a project falls behind schedule, or since the more organized programming mind will process forms more accurately, the data may be biased towards projects that are on time. This may lead to the unsubstantiated position that simply filling out of forms helps projects remain on schedule. While we believe that this may be true, it will be difficult to claim such interpretations.

In another experiment two parallel developments of a task were undertaken in order to compare one group not using any special methodology with a second group using a prespecified technique (Basili-Zelkowitz). One of the groups, knowing that they were being investigated but not exactly sure how, was extremely concerned about performance. They decided to change their normal behavior by using explicit requirements, good project control, top down development, etc. However, we wanted to monitor normal progress for this group, and this change was not their normal environment. Fortunately, the use of these "new ideas" lasted about a week and the group returned to its standard techniques and established methodology of programming. Since they were still defining and darifying the requirements for the task and had not begun any design, the impact on the entire project should be minimal.

The impact of the forms themselves is something which we would like to but cannot ignore. The contractor has asked for a los increase in costs just for filling out of this data. While we do not believe that such estimates are realistic, EASA has agreed to it for now in order to develop a valid data base.

Another problem with the forms was their generality. In order to have a standard data base, a general purpose set of forms has been developed. Unfortunately not every question is applicable in all situations. This adds overhead to filling out the forms since programmers do not know how to answer some of the questions. In our next iteration of the forms, we intend to abstract the typical answers being given at present and make the forms more of a checklist.

Given the completed forms, the next step was to encode them into a FDF11 data base. We chose to use the IMCRES relational data base system (Held) that runs under the UMIX operating system. This decision was mostly for its cost (\$75) compared to other relational data base systems (\$100,000); however, it did have a good reputation from other users. Important in any data base project is the need to get data into and cut of the system; however, with the large amount of data being collected, the tendency is to ignore output in an attempt to keep the input gates clear. This is clearly a danger, and we believe (so far successfully) that using a relational data base system allows for complex output queries to be easily written to solve this problem.

In our investigations of data base systems, INCRES was supposed to be relatively hard to install but was supposed to work well. After some initial tape problems and local hardware malfunctions, we found this to be true. We had little problems installing the system; however, in trying to adapt it to our use we did discover several limitations. INCRES is somewhat of a toy system and data relations must be kept somewhat simple (500 bytes and 49 fields per entry). In addition, the system uses many of the features of UNIX which makes it slow on our 64k EDFIL/45. This has partially restricted our activity but has not forced us to change our overall goals or procedures.

Once in the data base, the first task has been to preserve data validity. In addition, special programs have been written to check for missing data. For example, since forms are uniquely numbered, missing numbers are easily spotted and usually mean missing forms. Also some forms are to be filled out

weekly so it is easy to spot missing ones in that sequence.

#### DATA EVALUATION

The data evaluation phase consists of developing measures to evaluate the methodologies used on the projects. One technique recommended by the contractor was to provide instant feedback of the data on the forms. This was to help keep up the enthusiasm of the programmers for the entire endeavor. Since we did not want to report back any information that could be used for personnel evaluation, we designed some general purpose reports which give summary totals about each project.

The next step was to start developing more complex validity checks on the computerized data base. This has been the major emphasis of the laboratory for the last few months. Due to the large variation in programmer performance, we are trying as much as possible to make sure that what we have is correct. Comparing similar data in different ways on different forms is one way we are checking for consistency. For example programmer times from the resource summary and from the component status report can be compared. Also, error runs from the computer program run analysis form can be compared to the number of change report forms submitted. Looking at gross data on projects that have not been monitored is another technique. Also, interviews with some of the contractor personnel has been used to validate certain answers.

INCRES queries are now being written to extract detailed information from the encoded data. A plotting package has been implemented to plot project characteristics. We are currently at the stage where some of these queries are being tested on the data base.

### SUDGARY

As this report describes, the Software Engineering Laboratory has been in operation for a year and is clearly a very complex operation requiring constant

as they arise. More effort than we originally planned is going into day-today operation and maintenance. Forms and data base validity have become the critical section through which all laboratory activities now depend.

We have been developing a prototype laboratory which can be used to perform the needed experimentation in order to develop theories of program development. Although developed within the MASA framework for software development, we do not believe that the problems we encountered or our solutions to them are unique. Given such care in maintaining the data, we believe that the data which is being collected is valid, useful, and should lead to important insights about program development.

### RESPECTATION HERE

(Basili et. al.) Basili V., Zelkowitz M., et. al., The Software Engineering Laboratory, Technical Report TR-535, Computer Science, University of Maryland, May, 1977.

(Basili-Zelkowitz) Basili V., Zelkowitz M., Designing a software measurement experiment, Software Life Cycle Management Workshop, August, 1977.

(Held) Held G., Stonebraker M., Wong E., INGES- a relational data base system, Hational Computer Conference, 1975, pp. 409-416. SOFTWARE PSYCHOLOGY:

SHRINKING LIFE-CYLE COSTS

T. LOVE

General Electric Co Informations Systems Programs 1755 Jefferson Davis Highway Arlington, Virginia 22202 Early predictions that "thinking machines" would increase the unemployment rolls are being shown to be incorrect. In fact, a recent forecast suggests that "by 1985, automated information processing could be the most labor-intensive of all industries with the possible exception of agriculture". Who would have ever predicted that we may someday end up with more programmers than farmers?

In the military, government, and industry, the high cost of software is strictly a function of the labor involved in producing, operating, and maintaining code. Then once we have produced software by this labor-intensive and expensive process, frequently it does not do what it is supposed to do. Little wonder that when the "All Purpose Elixir" (structured programming) arrived on the market, lines formed in every direction to partake. However, as with most miracle cures, this one has not lived up to its initial advertising claims.

What the structured programming uproar may have done is to define the problem for the first time. That problem is that software problems are people-problems which cannot be solved by the simple addition of a faster black box on the computer system. This paper will describe an approach to solving these problems and describe some results from its application.

#### THE CASE FOR SCIENCE

Engineering methods demand some underlying set of rules and facts which can be followed in a disciplined way to produce the desired result. Discipline alone is really not sufficient. A group of engineers designing a bridge need to have the same theory of the effect of gravity and of the stress that the chosen materials will tolerate. Newtonian physics and carefully controlled stress experiments provide the basic rules which engineers use to design bridges so that they work the first time.

In software engineering, we do not know the underlying rules and facts that we need to do our work. Instead, we are busily imposing discipline and structure to our software production processes without any understanding why a particular discipline or structure should be preferable to another. Furthermore, the number of possible disciplines and structures is sufficiently large that evolution to better methods through trial and error may take centuries (assuming that we maintain careful records, which of course we do not)!

The ultimate success of software engineering will depend upon the development of the underlying scientific knowledge to answer our present-day "why" questions. Due to the inherent complexities built into these questions, it is clear that no one discipline alone can provide the answers. Some disciplines which antedate computers can be of significant benefit by at least providing some established methods for answering our questions, if not some relevant theories to guide our question asking. Two such disciplines which have been used in the research reported in this paper are psychology and statistics.

Therefore, improving software production and maintenance can be done in a rational systematic fashion using knowledge and technology which is well established. In this way software life-cycle costs can be reduced without reliance upon either our fallible intuitions or evolution. Some recent examples of this approach will be described below.

We are currently conducting experiments for the Office of Naval Research to examine four phases of the software life-cycle: (1) understanding, (2) modification, (3) debugging, and (4) construction. A preliminary experiment has been conducted to test various influences on the human understanding of software (Sheppard and Love, 1977).

In this experiment eight experienced programmers were asked to study FORTRAN programs and reproduce functionally equivalent programs from memory. Understanding was defined as the percent of functionally equivalent statements correctly recalled. The programs were written using three levels of control flow complexity and three levels of meaningfulness of variable names. The most meaningful variables had four to six characters, and the least mnemonic ones consisted of randomly chosen, single letters. All programs were written in standard FORTRAN, without any of the popular structured preprocessors. Meaningfulness of variable names was manipulated independently of control flow levels. The experimental design allowed each programmer to study and recall one version of each of three programs and all levels of the two primary independent variables.

An analysis of variance showed no significant effects due to mnemonic levels (see Figure 1). Curiously enough, the trend was toward improved performance with random letters -- exactly the opposite of our prior predictions! However, significant differences between control flow levels were found with the more structured programs being the easiest to recall.

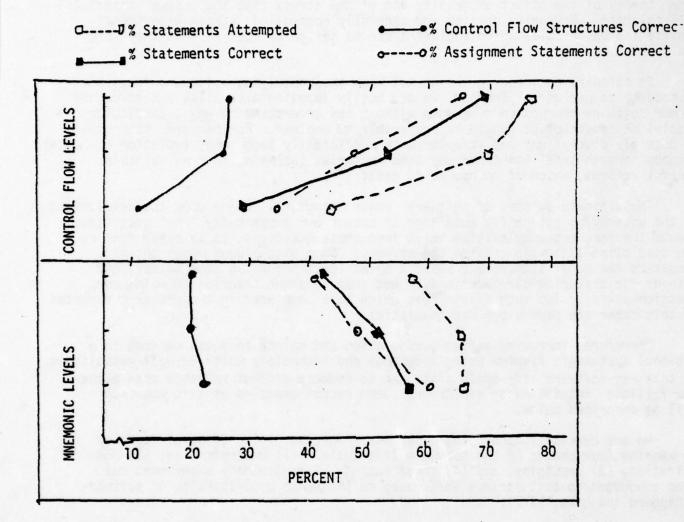


Figure 1: Analysis of Variance Results for Three Mnemonic Levels and Three Levels of Control Flow

Performance was also compared with Halstead's E, a measure of the effort required to write a program. A high correlation (-.81) was found. This experiment represents the first time that Halstead's theory of Software Science (1977) has been rigorously tested with proper experimental controls. Considering that we can account for more than 60% of the variance while ignoring differences in use of mnemonics and function of the program, suggests that our intuitions about programmer behavior may not be so accurate. In order to verify and gain greater confidence in our initial findings, a more extensive study involving 36 subjects and 81 programs is in progress. In this study, great care is being taken to use programs representative of those encountered by professional programmers. Also the experimental design will precisely determine all primary effects and interactions which are of interest.

In summary, GE is demonstrating that experimental studies of programmer behavior can be done and the nonintuitive findings of our research suggest that more experimental work should be performed. GE's work for ONR is unusual in the software research field in that it is testing precise predictions of a theory using controlled experiments.

#### INDIVIDUAL DIFFERENCES IN PROGRAMMER PRODUCTIVITY

Empirically, we have begun to accept the notion that two programmers with equal experience and credentials may program at radically different rates (up to 25 to 1). What we do not yet know is the distribution of these differences and why they are large. A field study will be described which addresses these questions.

In an academic introductory programming class, four generic types of information were obtained to answer these questions. This information included:

- Classroom performance measures (grades, test scores, etc.)
- 2. Unobtrusive measures from actual programs (number of runs, number of changes per run, type of change made to program, type of instruction changed, etc.)
- Questionnaires from each run (environment, expected time of completion of problem, source of assistance, type of error encountered, confidence that program will run, etc.)
- 4. Measures of human information processing (memory span, memory for programs, speed/accuracy of comparing digit strings, etc.)

The information was obtained from 3 sections of an introductory FORTRAN programming class taught in the engineering department at the University of Washington. Altogether, 633 runs from 54 students working on 4 programming problems were collected and analyzed. Each of the more than 40,000 source code instructions submitted for processing were collected and analyzed by a special purpose program modeled after one described by Nagy and Pennebaker (1974). In addition, another routine was developed to print a special purpose questionnaire on each output from each run to request the information described above.

Table 1 provides us with an initial overview of the student programming performance across sections and problems. There was a chasm between students' expectations regarding their programs and reality (see Figure 2). After 2 runs 82% of the students expected to have completed their programs, but in fact, only 21% did. After 6 runs, the comparative percentages were 91 and 71, and after 10 runs, 100 and 87. If such incredible inaccuracy exists for novice programmers, it is not surprising that professional programmers were unable to estimate time and costs of developing a program more accurately. To improve the accuracy of these estimations, some feedback system must be implemented. System-wide or userdeveloped systems to maintain run and cost data might be helpful. Interestingly, we observed that those students who performed well in the memory for programs experiment, as well as those who had higher scores on the digit span test, took fewer runs to complete their programming assignments. (see Table 2). There was also an inverse relationship between performance on the free recall learning task and the number of logical errors reported in programs. We have evidence here for a relationship between programming performance and human information processing ability, albeit complex!

One measure of student programmer performance obtained unobtrusively by the computer was the number of runs required to complete each programming problem. A stepwise regression analysis was done using the mean runs required by each student. for all problems as a dependent variable, and 18 other measures obtained from the questionnaires, the measures of human information processing, and score obtained in the course as independent variables. The results of this analysis are presented in Table 3 with the independent variables listed in the order in which they entered into regression equations,

From this first analysis, we see that score in class was not strongly related to performance as measured by number of runs, but seems related more strongly to the number of logical errors reported for a problem, digit span score, and program preparation time. Notice that the longer one spends designing a program, the more runs are required to complete the program; but the more time coding and keypunching a program, the fewer runs. Such a relationship might not exist for more complex problems, or more experienced programmers.

Another possible measure of programmer performance is the mean number of changes required per run of the program. This measure was a composite of the number of substitutions, deletions and insertions. Table 3 shows the results of the stepwise regression analysis using this measure as a dependent variable. We see that students with high scores on the perceptual speed test tended to make more changes per run than those with low scores on the test. In addition, the students who wrote larger programs tended to make more changes. The same analysis was repeated, using substitutions as the dependent variable, and a similar set of variables enters the equation. However, the coefficient of determination was somewhat higher than in the previous analysis.

Finally, which of these measures allows us to predict a person's grade in an introductory course? The answer is, very few. A stepwise regression was performed in which 18 measures of performance, human information processing, and data from the questionnaires could have been entered into the equation. Only two measures were significantly related to score in the classroom -- mean time to locate an error in a program and frequency of syntax errors. We can compare this relationship to that of GPA and the score in the course. For the 27 students who agreed for us to obtain their overall college GPA, the correlation between GPA and course grade

TABLE 1

			MANADY OF DEREN	CHAMADY OF DEDECTORANCE MEACIDES			
PROBLEM	SECTION	1 37	X CHANGES/RUN	XX CHANGES/RUN	X RUNS	TOTAL RUNS	NO. OF STUDENTS
-	Y	78.5	8.75	0.11	9.65	193	50
. 2	6	49.6	2.43	4.00	4.63	74	
9	<	64.9	.80.9	9.00	6.71	14	71
		62.1	5.42	9.00	4.80	72	15
	v	51.0	5.98	12.00	6.23	æ	<b>E</b>
4	4	79.6	9.08	11.00	3.71	52	14
	•	79.6	12.54	16.00	3.92	47	12
OVERA	OVERALL Ks	67.7	7.12	10.00	5.92	633	
8		<b>A</b>	= SOFTWARE EN	(A = SOFTWARE ENGINEERING SECTION)	(NO)		

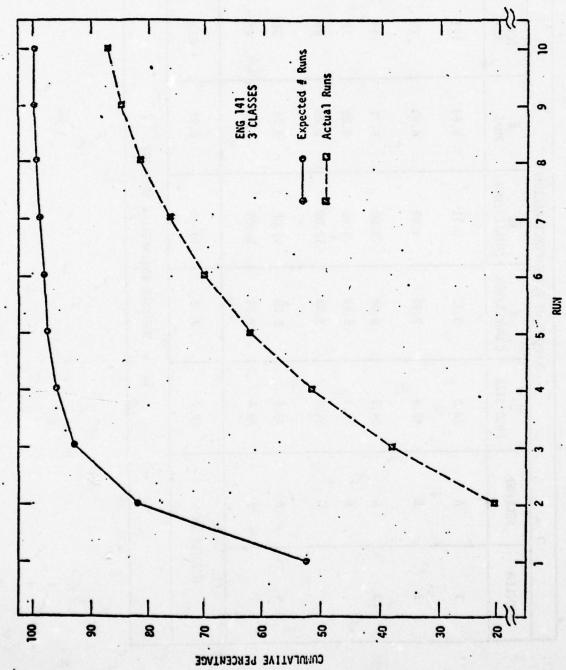


Figure 2 A Comparison of Actual to Expected Number of Runs Required to Complete Computer Problems

.[]

CORRELATION	HIP MEASURE	PROGRAMING PERFORMANCE MEASURE
*89*	PERCEPTUAL SPEED SCORE	NUMBER SUESTITUTIONS
62*	PERCENT LINES RECALLED (Memory for Programs)	MAXIMUM RUNS
*25*	NUMBER CORRECT CPAL	NUMBER SUBSTITUTIONS
.57*	PERCEPTUAL SPEED SCORE	TOTAL PROGRAM CHANGES
52*	DIGIT SPAN SCORE	. MAXIMUM RUMS
.51*	PERCENT LINES RECALLED (Hemory for Programs)	TIME TO LOCATE ERROR
45*	PERCENT CORRECT FREE- RECALL LEARNING	NUMBER LOGICAL ERRORS

\* p(a) < .05; two tailed test

Significant correlations between measures of human information processing and computer programming performance.

TABLE 3

DEPENDENT VARIABLES	BETA	INDEPENDENT VARIABLES	SIMPLE	R <sup>2</sup>	SIGH.
Runs .	. 0.50	Logical errors	0.62		
	-0.45	Digit span score	-0.52		
•	-0.57	Time to prepare programs for submission	-0.50		
	-0.47	Syntax errors	0.17		
	0.41	Time to design programs	0.26	0.85	0.01
Mean changes per	0.70	Perceptual speed score	0.56		
run	0.33	Program size	0.29		
	0.32	Confidence of running	-0.02		
	. 0.21	Time to locate errors	0.22	0.53	0.01
Mean substitutions	0.57	Perceptual speed score	0.68		
	0.31	CPAL score	0.56		
	0.28	Program Size	0.17		
	0.74	Confidence of running	-0.11	0.60	0.01
Score in course	-0.43	Time to locate errors	-0.42		
	-0.25	Frequency of syntax errors	-0.24	0.24	0.05
Logical errors	0.62	Runs ·	0.62		
<b>&gt;</b>	0.26	Time of day	0.32		
	0.13	Digit span score	-0.17		
	-0.28	Perceptual speed score	-0.20		
	0.27	CPAL score	0.15	0.54	0.01

(Minimum sample size for any pair of correlations = 29)

Global Stepwise Regression Predicting Programming Performance

was .58 (coefficient of determination = .34). Therefore, course grade seems to measure test-taking ability more than it measures programming performance as defined here.

The low correlation between grade and programming performance may not imply that current methods of assigning grades are invalid, but, rather, it may simply suggest that programming performance is difficult to measure objectively. A priori, we assumed that "A" students would be able to write correct programs, keypunch them properly, get them working in one or two runs, and make few, if any, changes. An alternative hypothesis is that there are two types of "A" students — those who are able to develop a correct algorithm and implement it without making any logical changes and "A" students who have both algorithm and implementation difficulties, but persist until the problem is solved correctly. "A" student type I would clearly be the more desirable programmer for any organization.

Referring back to Table 3 we can see the results of a stepwise regression analysis where mean logical errors per problem was used as the dependent measure. The first two measures might have been expected; viz, the more runs a person takes, the more logical errors. The next three measures are measures of human information processing ability and their presence in this equation is sufficiently interesting to merit an independent series of studies. The relationship between these measures of human information processing and logical errors is not that strong, but reliable information which helps us eliminate logical errors in computer programs is extremely valuable. If such a relationship were substantiated in further studies, there are at least two distinct benefits. New programmer selection procedures could be developed to select programmers who have good short term memories and programming techniques could be modified to minimize short term memory demands.

In the stepwise regression just described, variables were entered into the regression equation according to their individual ability to improve the prediction of the dependent variable. A similar analysis can be done in which variables are forced into the equation according to some predetermined ordering. Table 4 shows such an analysis in which programming performance measures were predicted by (1) measures of human information processing, (2) class grade for the quarter, and (3) the subjective measures obtained by the questionnaire. The sizes of the coefficients of determination are sufficiently large to add a sizeable measure of support to our hypothesized relationship between short-term memory ability and programming performance.

In closing, our major findings will be reiterated.

- Students took a sizeable amount of time to use the FORTRAN language to solve simple problems. Particular difficulty was experienced with input/ output statements.
- There was some evidence that students may take a troubleshooter approach
  to debugging their programs -- find the obvious error, correct it, and
  try again.
- There was a chasm between students' expectations of the time required to complete a programming project and reality.
- 4. There was no significant correlation between the number of runs required to complete a programming project and classroom performance.

DEPENDENT VARIABLE	INDEPENDENT VARIABLES	R <sup>2</sup>	△ R <sup>2</sup>
KEEN RUNS	HIP	. 14.	.41
	CLASS GRADE	.41	.00
	SUBJECTIVE FACTORS	1.00	.59
MEAN CHANGES PER RUN	HIP	.35	.35
	CLASS GRADE	.36	10.
	. SUBJECTIVE FACTORS	.59	.23
MEAN SUBSTITUTIONS	HIP	.53	.53
9503 9503 9503 9503 9503 9503 9503 9503	CLASS GRADE	.53	00.
	SUBJECTIVE FACTORS	89.	.15
MEZH LOGICAL ERRORS	нтр		.24
	CLASS GRADE	. 25	6.
	SUBJECTIVE FACTORS	29.	.42
1			

Hierarchical Regression Predicting Program's Performance

posterior a

 Analyses and inspection of the data suggested that a relationship may exist between logical errors in programs and the short-term memory ability of programmers.

#### CRITICAL FACTORS IN SOFTWARE DEVELOPMENT

Scanning any recent publication in Software Engineering, one discovers a number of suggestions for improving the software production and maintenance processes. These suggestions address a number of problems which are presumed to be critical to the production of software. But nowhere is there a list of the most critical factors ranked in some order of importance. We at GE/ISP have recently completed an effort to provide exactly this information (Love and Fitzsimmons, 1977).

We designed a three part survey to send to software developers within GE. In the first section, we presented a list of 100 difficulties likely to be encountered during a software development effort and requested the respondent to rate the criticality of each difficulty. In the second section, we requested a ranking of six categories of difficulties to determine which were most critical. Then in the final section we used a series of open-ended questions to determine which difficulties were considered most important.

Of the 200 surveys distributed, 89 were received from 4 GE locations. The respondents, all software developers, represent a range of experience with respect to job positions, type of programming done, number of years of experience in software development, and programming languages used. They also represent a cross section of software developers throughout the country, with 24 from Arlington, VA; 37 from Sunnyvale, CA; 25 from Syracuse, NY; and 3 from Schenectady, NY.

#### Section I

Thirteen of the 100 questions had an average criticality greater than or equal to 5.0 (or very critical). (see Table 5). Of these 13, 6 were from the Management Plan subgroup, 4 from the Requirements Analysis, and one each from Environment, Development System, and Operational System. No questions from the People subgroup were above this break point.

While inspecting the set of most critical questions in each subgroup, we discovered some interesting similarities. For the Requirements Analysis, we find that the most critical aspect is that the requirements be complete and well specified. Also, the procedure for clarifying the requirements was considered critical.

With regard to the Management Plan subgroup, all of the critical questions involve the estimation and allocation of appropriate resources to the software development team. This is a critical need of software managers which is receiving little attention. While a plethora of so-called software tools are being used to help the analyst and coders, no such set of tools is being used by software managers. Also software developers want more access to more reliable software development systems. Oversimplifying, software developers want to know what to do and they want sufficient resources to do it (including calendar time, skilled personnel, and computers).

The objective of another analysis of the data from Section I was to investigate the relationship among the 100 questions. We essentially asked if we could have employed a small set of ratings to obtain the same information. For this analysis

Table 5. High Outliers (Questions Rated ≥ 5.0)
Section I

REQUIREMENTS ANALYSIS	Q1 Q2 Q12 Q13	Poor description of overall purpose of the software Lack of detailed specification of the problem Major aspects of the requirements not included Inadequate procedure for clarifying the requirements
MANAGEMENT PLAN	Q14 Q23 Q25 Q30 Q31 Q37	Ineffective project management Poor allocation and management of resources Failure of management to provide for sufficient skills and talents to accomplish the task Underestimate of the resources required to accomplish the task Underestimate of the difficulties which arise due to the size of the project Overly optimistic completion schedule
ENVIRONMENT	Q46	Insufficient access to computer
DEVELOPMENT SYSTEM	Q63	Unreliable development operating system
OPERATIONAL SYSTEM	Q100	Lack of time for initial testing of operating system

## Low Outliers (Questions Rated ≤ 3.0) Section I

ENVIRONMENT	Q45	Difficult to obtain general office supplies
DEVELOPMENT SYSTEM	Q75 Q77	Inappropriate identation of code makes it more difficult to understand Variables within the program tend to be global vs. local

we performed six separate factor analyses, one on each logical subset of the ratings. Regrettably, we could not perform a single analysis on all 100 ratings because only 89 surveys were ultimately returned -- far less than the 300 to 500 required.

Briefly, factor analysis is a method for determining  $\underline{k}$  underlying factors from  $\underline{n}$  measures. These factors, derived from the intercorrelations among the  $\underline{n}$  measures, are hypothetical constructs or variables that are assumed to underlie the original measures. The result of such an analysis is not a verbal description of the underlying factors. Rather, it is a matrix of factor loadings, ranging from -1.00 to +1.00, which express the relationships between the original measures and the derived factors. To the extent that a measure relates to a factor it is said to be loaded on that factor. However, understanding and describing the factor loadings which result from such an analysis is a subjective exercise guided by imprecise rules.

In our factor analyses we were seeking to identify the primary underlying factors which are related, either positively <u>or</u> negatively, to the success of a software development project. The results of the factor analysis can also be used to estimate the criticality of the hypothetical factors so derived.

Table 6 lists the 23 factors identified in the factor analysis, ranging from most critical to least critical. A larger survey of this type would reveal an even smaller set of factors.

## Section II

Data from 78 surveys were used to obtain the total number and proportion of votes for each paired comparison. These data were then used to obtain (1) the total percentage of votes for each of the six variables and (2) the average proportion of votes for each variable with standard deviations (see Table 7). The average proportion of votes is an interval measure which implies that we can meaningfully interpret the differences between the variables as well as the ordering. Clearly the Requirements Analysis and Development System are the critical variables according to this analysis. At the other extreme, the work Environment seems unimportant to software practitioners.

## Section III

The final three questions requested subjective responses concerning the respondent's own difficulties and successes in past software development projects. Responses were categorized into <u>classes</u> of difficulties reflecting the six variables used in the multiple ranking exercise. For example, both of the following responses were categorized under "Poor Management":

"Management refuses to make decisions (buck-passing, finger-pointing)."
"Management refuses to listen to and accept ideas of software developers."

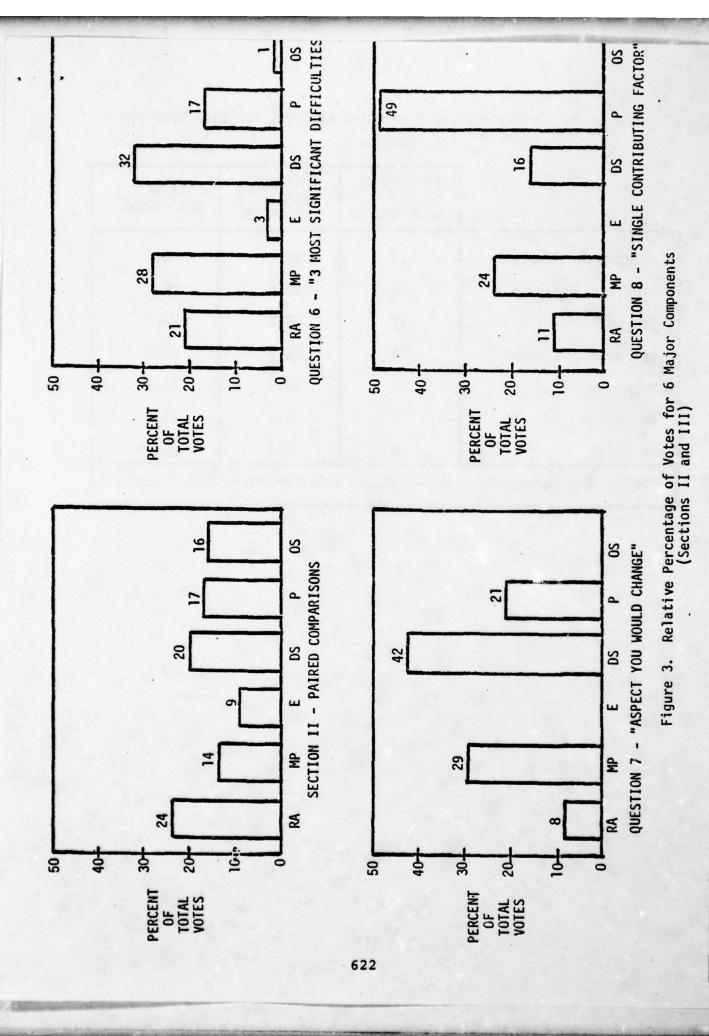
When possible, each class is broken down to show the major points mentioned in each.

Following is a comparison of the results of Section II, the paired comparisons measure, and the responses to Questions 6, 7, and 8 in Section III (see Figure 3). This comparison shows the percentage of votes (or times mentioned) for each of the major components of the software development system.

FACTOR	MEAN CRITICALITY
Poor project estimation	. 5.42
Lack of detailed specification of the problem	. 5.33
Conflicting and incomplete requirements specifications	. 5.09
Poor reliability of development system	. 4.81
Difficult-to-use computer systems	. 4.78
Difficulties using operational system	. 4.48
Poor diagnostics from operational system	. 4.47
Ineffective personal ability of manager	. 4.46
Ambiguous team structure	. 4.41
Inappropriate structure of software team	. 4.32
Inadequate training and review of team members	. 4.22
Distracting and uncomfortable physical working environment	. 4.14
Difficulties with operations personnel	. 4.13
Inadequate technical reference material	4.13
Complexity of code makes program difficult to understand	. 4.03
Failure to properly manage development team	. 3.96
Difficult-to-work-with co-workers	. 3.93
Requirements difficult to understand	. 3.80
Poor programming style	. 3.67
Poor relationships among co-workers	3.65
Unspecified software standards	. 3.64
Inadequate software progress monitoring	
Unresponsive support services	3.38

Table 7. Percentage and Proportion of Votes for Each Variable Section II

	PERCENTAGE OF TOTAL VOTES	AVERAGE PROPORTION OF VOTES	STANDARD DEVIATIONS
Requirements Analysis	24	84.4	9.5
Development System	20	68.8	26.7
People	17	58.2	26.5
Operational System	16	55.2	28.8
Management Plan	14	50.0	31.9
Environment	9	32.5	33.9
GE Stock	0	0.0	0.4*



Personal - Personal

E Comment

-

The two categories of difficulties which were judged most critical in Section I were Management Plan and Requirements Analysis. By contrast, in Section II we found that Management Plan was ranked fifth with only 14 percent of the total votes. But in Section III, where we categorized the responses into the appropriate categories, we again see Management Plan being mentioned second most frequently. This suggests that respondents did not properly understand what was implied by the term Management Plan in Section II but did consider it quite important when considering individual aspects of it.

We also see variation in the relative position of Requirements Analysis. It is considered quite critical in Sections I and II but, in the series of questions in Section III, interesting variability exists. Although Requirements Analysis gets 21 percent of the votes when asked for the "most significant difficulties", it gets only 8 percent when asked "What aspect should be changed?" This discrepancy suggests that even though software developers consider the Requirements Analysis important to the success of a project, it is perceived as an inherently difficult problem for which better procedures are not likely to be developed.

Another variation in relative positions can be seen in the People subgroup. When asked "What was the most important single factor contributing to high-quality software?", respondents listed competent and cooperative people 48 percent of the time. However, it was mentioned less than half as many times when asked "What aspect of your current software development process would you change?" Moreover, the People were mentioned only one-third as many times when asked for the "most significant difficulties". This may indicate that while the People are not the main source of difficulties, competent software developers are perceived as the best solution for overcoming these difficulties.

We also see that the Development System was ranked first when asked for "the most significant difficulties" and "What aspect would you change?", with 32 percent and 42 percent of the votes, respectively. On the other hand, the Development System was the "single contributing factor to high-quality software" only 16 percent of the time. This suggests that software developers are having a significant amount of productivity difficulty with the Development System, a problem which they feel could and should be dealt with.

We think of this survey as a pilot study before a more sizeable survey effort is done. With as many as 1,000 respondents, we could perform deeper analyses which might prove to be quite revealing. For example, how different are the problems in organizations developing different types of software working in different physical environments? Are there consistent differences as a function of the type of development system are being used? Do standards really seem to impact the attitudes or performance of the software developers?

Another distinct value of surveys is to measure changes in attitudes as a function of changes in operational procedures. Much effort is currently being expended to impose so-called structured programming techniques throughout the software industry, including GE. Based upon the results of this survey, this type of change is not likely to have great impact on employee attitude. But, to measure the true impact of procedure changes, we must begin to collect data on employee attitudes and employee performance. The determinants of programming performance may be quite different from those that the programmers identify.

#### CONCLUSIONS

The production of software involves a complex system which is not only difficult to study but also difficult to understand. Because of the inherent complexities in such a system, changing just one component in it cannot be done in isolation. Any change is going to perpetuate further changes -- some predictable and some quite unpredicatable, some beneficial and some unbeneficial. In order to tune the system to make it more efficient in a predictable fashion, a scientific approach is essential. The major components of the system need to be monitored over time to determine the precise effects of perturbations in the overall system. By properly combining this monitoring with basic research, software production can be transformed into a predictable process.

Once the system is understood and sufficient monitoring data have been scrutinized, we will be better able to predict and control the current high costs associated with software production and maintenance.

#### REFERENCES

- Halstead, M.H., <u>Elements of Software Science</u>, Elsevier North-Holland, Inc. 1977
- Love, T. & Fitzsimmons, A.B., A Survey of Software Practitioners To Identify Critical Factors in the Software Development Process, Technical Report Number TIS 77ISP002, 1977
- Nagy, G. & Pennebaker, M.C. A step toward automatic analysis of student programming errors in a batch environment. <u>International Journal of Man-Machine</u> Studies, 1974, <u>6</u>, 563-578.
- Sheppard, S.B. and Love, L.T., Testing Influences on Human Understanding of Software, General Electric Co.

#### COMPUTER SYSTEMS ENGINEERING MANAGEMENT EDUCATION

Dr. John H. Manley

Assistant to the Director The Johns Hopkins University Applied Physics Laboratory Laurel, Maryland 20810

Presented at the U.S. Army Computer Systems Command Software Life Cycle Management (SLCM) Workshop, Airlie House, Warrenton, Virginia, 22-23 August 1977. Copyright, Dr. John H. Manley.

## ABSTRACT

#### COMPUTER SYSTEMS ENGINEERING MANAGEMENT EDUCATION

Dr. John H. Manley

Assistant to the Director The Johns Hopkins University Applied Physics Laboratory

The advent of microprocessors, communication satellities, on-line systems and a wide variety of embedded computer systems has increased the depth of technical competence required to develop and integrate automated system components. The trend toward system development using building blocks (modules) possessing "standard" interfaces has permitted the emerging field of software engineering to grow explosively. The new "software engineer" no longer requires detailed knowledge of the internal logic of most of the growing number of software modules he uses to develop increasingly complex systems. On the other hand, computer system engineering managers need specialized assistance from a wider variety of technologists in this new environment to help them make decisions such as selecting a "best" set of modules, estimating system costs, determining work breakdowns, developing training programs, and so forth.

Thus, technological change has created and is continuing to expand three somewhat new and separate career fields in the computing world:

- (1) Computer system engineering management,
- (2) computer system engineering and maintenance, and
- (3) computer system resource engineering.

These occupational modifications, together with newly defined divisions of labor, have caused many of our traditional education programs in the computing world to become obsolescent, especially in the area of management. It is well known that many leading universities have addressed the software engineering problem and have made great progress in this area. The need for a more "systems oriented" and an "engineering-type" approach to building automated systems is also well known. Formal university-level training of computer system engineering managers, on the other hand, is relatively new.

At The Johns Hopkins University, courses in "computer systems engineering management" for graduate students in Electrical Engineering and Computer Science have been developed and taught for the first time in the 1976-77 school year. The two-semester series has also been summarized and taught in a 1977 summer session computer science survey course. The distinctive approach at Hopkins involves a blending of science and management through modeling and student onthe-job experimentation. A special effort is made to differentiate between "management style" and "management science". The program is oriented to first provide the student with a global perspective of computer system management and what top-level policymakers desire, e.g., DoDD 5000.29. A primer on management is provided to show what is "style" and what is not.

Through various projects and classroom exercises, the differences in management thinking and technologist thinking is explained based upon the physiological fact of the lateralization of the human brain. Using this foundation, a "bottom up" approach to instilling discipline in the computer system engineering management world is then taught using a "General Procedural Model." Finally, the student attempts to practice what has been preached by making a real management improvement in his own organization.

The courses blend real world management with management science, behavioral science, "pure" scientific method, engineering methodologies, and most important of all, common sense. The bottom line is to increase student awareness of the management world and provide them with a realistic methodology for improving that world on a day-to-day basis.

This presentation will provide a summary of the techniques used in the Johns Hopkins educational program to generate discussion and develop new research initiatives in this area.

## PRESENTATION OUTLINE

Computer Systems Engineering Management Education
The Johns Hopkins University

Dr. John H. Manley

#### Background

Historical Problems
Current Solutions

Continuing Problems

## Requirements for A More Permanent Solution

Differentiation Between Policymaking and Practice
Differentiation Between Top-Down and Bottom-Up Change
Improved Memory Structures for All
Broadened Knowledge Base
Change Implementation Skill Development
New Research Initiatives

### Academic Subjects

Computer System Engineering
Project Management
Behavioral Science
Decision Science
Management Science

Scientific Method

Model Building

Change Implementation

Oral and Written Communication

Classroom Exercises

Interviewing Techniques

Communication Techniques

Peer Pressure

Information Pressure

Time Pressure

Major Project

Identify a "Real World" Problem

Find a Solution

"Sell" the Solution

Implement the Solution

Report on the Process

Overcome Pressures of:

Time

Peers

Supervisors

Academic Requirements

Bottom Line

Restructure the Student's Thought Process

Learn to Use "Right-Brain" to Communicate With Management
Return to "Left-Brain" When Appropriate

Promote "Bottom-Up" Change Consistent With "Top-Down" Policies
Keep Focus on Computer Systems as a Functional Area

Francis .

# A THEORY OF THE MARKET DEMAND FOR INFORMATION ANALYSIS CENTER SERVICES 1, 2

Peter G. Sassone<sup>3</sup>

The purpose of theory is to explain observations by showing that the observations are analogous to or special cases of known and predictable relationships. Just as the heliocentric theory of the solar system "explained" observations on planet movement by demonstrating the movement was predicted by the known laws of terrestrial physics, and the first theories of gas behavior "explained" by positing an analogy between gas molecules and solid vibrating spheres obeying the laws of mechanics, a theory of demand for Information Analysis Center (IAC) services should explain, or account for, observations on demand by showing they are consistent with accepted economic theory. (One usually prefers to speak of accepted theory rather than known laws in economics.) The purpose of this paper, then, is to develop a theory of the demand for IAC services — a theory consistent with accepted economic theory and consistent with the observations on demand for these services.

An Information Analysis Center is a formally structured organizational unit specifically but not necessarily exclusively established for the purpose of acquiring, selecting, storing, retrieving, evaluating, analyzing, and synthesizing a body of information and/or data in a clearly defined specialized field or pertaining to a specific mission with the intent of compiling, digesting, repackaging, or otherwise organizing and presenting pertinent information and/or data in a form most authoritative, timely, and useful to a society of peers and management.

This research was partially supported by the National Science Foundation's Division of Science Information by Grant OSI75-12741 (formerly SIS75-12741) to Metrics, Inc. The author acknowledges the helpful comments of the Project Overview Committee and the staff of Metrics. Of course, responsibility for all errors rests with the author.

<sup>&</sup>lt;sup>3</sup>Associate Professor of Economics, Georgia Institute of Technology and Research Associate, Metrics, Inc.

For concreteness, this paper will concentrate on analyzing demand for two specific IAC services: providing a handbook and providing inquiry response. The model and results are generalizable, with some loss of tractability, to the provision of many other information services. The observations to be explained may be cast as "stylized facts," to borrow a term from the economic growth theory literature. These stylized facts are:

- Low willingness-to-pay for IAC inquiry response by potential and actual users.
- 2. High elasticity of demand for inquiry response.
- 3. Low volume of service actually demanded from IAC's.
- 4. High value of the information provided by the IAC to its users.
- Adverse effects of handbook sales on demand for inquiry response.

Facts 1, 2, and 3 can all be considered as reflecting a lackluster market demand for IAC inquiry response. Fact 4 gives rise to a paradox (and the need for explanation) since, typically, the information provided by an IAC inquiry response service is very important to the customer. Presumably, he should be willing to pay quite a bit to get the information, and his demand should be relatively inelastic. The fifth fact presents no paradox, but is simply an observation which a successful theory should be capable of accounting form.

Since an IAC provides information <u>services</u>, it is important to recognize that it is the value of their service, not the value of the information, which determines the economic viability of an IAC. This distinction is important as long as the IAC does not occupy a monopolist's position with respect to any types of information. To see this, consider the analogous problem of valuing the services of a surgeon who performs lifesaving operations. Is the value of the <u>surgery</u> the value of the life saved? Yes\*. Is the value of the surgeon's <u>service</u> the value of the

1

<sup>\*</sup>Let's grant the point, for argument's sake.

life saved? No! This is simply because there exist a large number of surgeons with comparable skill. The lifesaving surgery could be provided by a large number of service (surgery) providers. The value, to the patient, of that surgeon's service is the cost of another equally skilled surgeon's service. If all surgeons (assumed equally capable) charged \$1000 for a particular operation, the value of any one surgeon's service is only \$1000, since that is the cost of acquiring a perfect substitute. The value of the service of a particular supplier and the value of the service supplied are vastly different concepts. The same point can be made, perhaps more clearly, with a physical commodity rather than with a service. Consider the commodity, aspirin. In Figure 5, assume D is the market demand for bottles of aspirin. If aspirin were supplied by a perfectly discriminating monopolist, he could charge each consumer the total of what each consumer values the aspirin. Thus, the first bottle could be sold at P1, the second at P2, etc. The monopolist is able to sell his provision of the good at the value of the good itself to each particular buyer. This situation can be contrasted with a competitive market in which many firms provide a homogeneous good (each firm's product is indistinguishable from that of any other firm), and no one of them provides a significant amount of the total. Let P' be the price which prevails under this case. Each supplier can sell his provision of the good at a price no higher than other suppliers sell their provision. For to set a higher price would be to lose all sales, since no buyer will willingly buy the same good at a higher price. Thus, the price which can be charged in the competitive market does not depend on the value of the good itself, but on the prices other suppliers set.

The upshot of the foregoing is that insofar as IAC's are <u>not</u> the sole providers of information, the prices buyers of information are willing to pay the IAC do not depend on the intrinsic value of information, but on the prices other information suppliers have established. The point is stressed here because it is so easy to lapse into the mistaken notion that the value of an IAC is the value of the information it provides. Rather, its value is the cost of the best alternative information source.

Only in the instance that an IAC provides totally unique information can the IAC's value be measured by the value of the information (i.e., the IAC is a monopolist).

As a first approximation, imagine that all questions ask for the numerical value of some parameter. Parameters might include physical constants, economic values, production levels, dates, etc. Assume a measurable and known cost to the user of acquiring each item of information. Likewise, assume a measurable and known monetary benefit to the user associated with each item. The rational user will determine the net benefit of each item (benefit less cost) and purchase the information item as long as net benefits are positive. The whole set of information items may be arranged according to diminishing net benefit, as in Figure 1. The curve is interpreted as follows: the  $\mathbf{q}_0$  information item yields net dollar benefits of  $\mathbf{v}_0$ .  $\mathbf{q}^*$  is the quantity purchased by the rational user since each unit up to  $\mathbf{q}^*$  adds more to net benefits and each unit beyond  $\mathbf{q}^*$  detracts from net benefits.

Some reflection will reveal that the graph of benefits and the graph of costs of each information item (arranged along the horizontal axis in the same order as in Figure 1) will not, in general, be representable by a smooth curve. For the (gross) benefit of any item may differ markedly from the (gross) benefit of an adjoining item, and likewise for costs. All that is known for sure is that the <u>differences</u> in benefits and costs decline as one moves to the right in Figure 1.

Figure 2 illustrates benefit and cost curves which could give rise to a net benefit curve like Figure 1. So far, no mention has been made of the IAC. In particular, the cost curve in Figure 2 (and thus implicit in Figure 1) represent the costs in the <u>absence</u> of the IAC. Now suppose the IAC can provide information to the user at a constant unit cost, say c. If c were less than  $C_L$  - meaning that the IAC could always provide information at a lower cost than any other information service (where providing information to oneself is one of those "other" information services), the user would purchase <u>all</u> information from the IAC (case  $C_1$ ). On the other hand, if c were greater than  $C_H$ , the user would purchase no

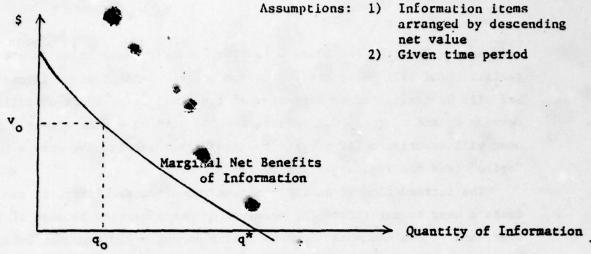


Figure 1. Net Value of Information, Items arranged by descending net value.

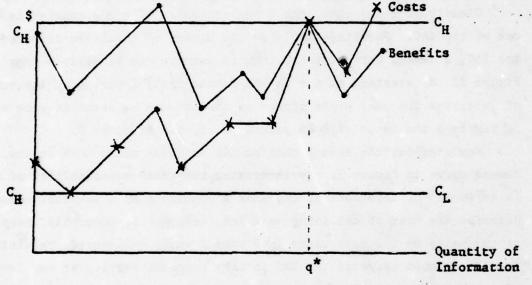


Figure 2.

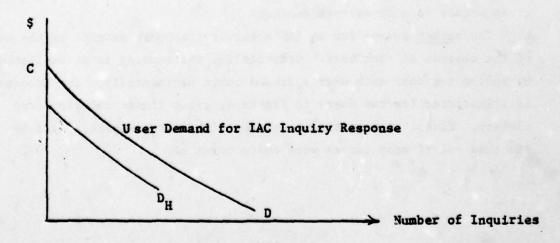


Figure 3. User Demand for IAC Inquiry Response

information from the IAC (case  $C_2$ ). The first of these cases is not realistic and will be ignored. The second case is of limited interest, and will be considered an extension of the final case, where ca falls between  $C_L$  and  $C_H$  (case  $C_3$ ). Here, the IAC acts as a peak shaver – the user will substitute IAC service for some other service whenever a cost "spike" (see the figure) is cut by the IAC cost, c.

The introduction of an IAC can have two effects. First, it can cause a user to substitute IAC service for other service because of lower cost, and it can cause an increase in the amount of information demanded, because benefits may exceed c beyond  $q^*$ .

Clearly, for any user, the lower the value of c the greater is his use of the IAC. Focusing simply on the number of inquiries directed to the IAC, a demand curve for IAC inquiry service can be derived from Figure 2. By starting at  $c = c_H$  and successively lowering c, the number of inquiries the user would direct to the IAC can be noted at each c. In Figure 3 the locus of such points is plotted as curve D.

Now consider the effect that an IAC handbook would have on the demand curve in Figure 3. By increasing the ready accessibility of a large amount of information, the user's ownership of a handbook would decrease the cost of obtaining some information. Consequently, many of the peaks on the cost curve in Figure 2 would be lowered, resulting in a diminished usage of the IAC inquiry response service at any level of c. This has the effect of shifting the curve in Figure 3 to the left. The new curve is represented as D<sub>H</sub>, the demand for inquiry service given access to a substitute handbook.

The <u>market</u> demand for an IAC's inquiry response service is the sum of the demands of each user. Graphically, the summing is accomplished by adding together each user's demand curve horizontally. The process is illustrated for two users in Figure 4, using linear relations for clarity. Kinks, such as the one in the rightmost quadrant, would be smoothed out if many curves were added together.

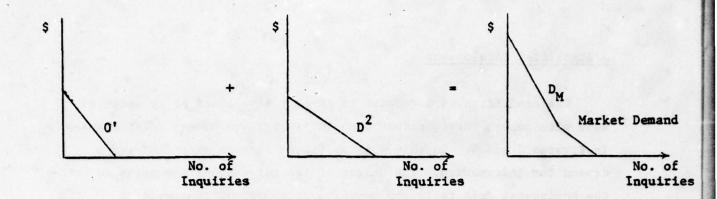


Figure 4. Illustration of how Market Demand is Determined from Individual Demands

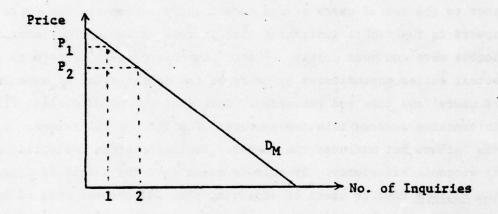


Figure 5. Smoothed Market Demand Illustration

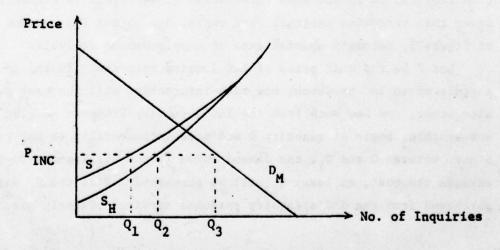


Figure 6. Market Supply and Demand Model for Inquiries

#### A Simplified Development

The exposition of our model is greatly simplified if we agree to work with smooth curves rather than the jagged and kinked relationships in Figures 2 and 4. In this spirit, Figure 5 represents the user's demand for information. The height of the curve above any point on the horizontal axis tells the benefit, or value, of that unit of information (response to an inquiry) to a user. In Figure 6, S is the marginal-cost-of-information curve, excluding consideration of the IAC. The height of the curve at any level of inquiries tells the incremental cost to the set of users of one more inquiry response. The curve slopes upward to the right, indicating that increasing amounts of information become more and more costly. "Cost," in this context, refers to either actual dollar expenditures by users or the dollar value of expenditures of users' own time and resources. Note that our model divides all information sources into two sectors: The IAC and all others. A includes the latter, but excludes the former. Implicit in the definition of S is economic efficiency. By this is meant that the height of S represents the minimum cost to users of acquiring that incremental unit of information. Thus, S is constructed assuming the least costly source is used for supplying each unit of information service. Su is the same supply curve, but shifted downward in the event the IAC provides a user handbook. The handbook, of course, lowers user information costs by reducing the time required to locate some information items. This is reflected in the lower than otherwise marginal cost curve, S<sub>H</sub>. S and S<sub>H</sub>, coupled with D of Figure 5, become a special case of supply-demand analysis.

Let P be the unit price of IAC inquiry response. First, in the absence of an IAC handbook, how much information will the user purchase altogether, and how much from the IAC's inquiry response service? To answer this, begin at quantity 0 and move incrementally to the right. Since, between 0 and  $Q_1$ , the demand curve (hence willingness-to-pay) exceeds the cost, at least  $Q_1$  will be purchased. Will the  $Q_1$  units be purchased from the IAC's inquiry response service? Clearly not, since

each unit costs less on S than on P. However, between Q1 and Q3, PIAC is less than S, and P is also less than D. Therefore, the user will purchase  $Q_3 - Q_1$ , but from the IAC. To the right of  $Q_3$ , willingnessto-pay is less than cost (i.e., D curve is below P line), so no purchase is made. Thus, of a total purchase of  $Q_3$ , only  $Q_3 - Q_1$  is purchased from the IAC. The remaining Q1 units are purchased from the least expensive "other" suppliers of the information service. As noted above, this can include self-supply. At this point the effect of the handbook is evident. With  $S_H$  the supply curve, only  $Q_3 - Q_2$  (in addition to the handbook) would be purchased from the IAC. The model suggests that the effect of a handbook is to reduce inquiries, as has been apparently observed. Figure 6 can be used to derive the demand curve for IAC inquiry response. This is done by varying P, and noting the change in quantity of inquiry response demanded. Thus, using S as the supply curve, if P were slightly higher, less than Q3 - Q1 would be demanded. Indeed, quantity demanded goes to O at a price equal to the intersection of S and D. Some reflection indicates that the demand for inquiry response is given by the difference between D and S, as long as D - S is positive. Figure 7 illustrates this demand, denoted DTAC. Note it is uniformly lower than D, reflecting the point made earlier about the difference between the value of information and the value of an IAC's providing information. Also, since the observed point on D will be right of the observed point on D, the demand for inquiry response will be more elastic than the demand for information. Finally, note that if Su applies in Figure 6, Du applies in Figure 7.

This model of market demand for IAC inquiry response, summarized by Figures 2, 3, 4 or 5, 6, 7 (for the "smooth" case), explains the observations set out earlier. In particular, the concept of a user's own supply of information curve permits the paradox of high information value - low demand for information service to be resolved. While the information has high value, the providing of the information has value limited by the relative cost to the user of competing services, as reflected in the users' supply curve. Further, it is this curve which leads to the difference in elasticities between total information demand and IAC inquiry response demand.

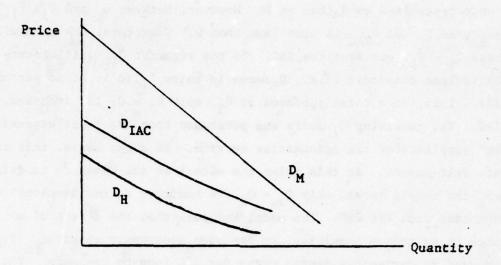


Figure 7. Demand for IAC Inquiry Response Compared to Demand for all Information

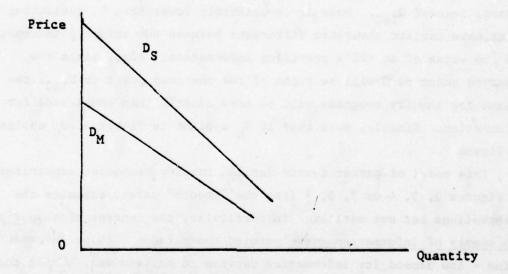


Figure 8. The Market and Social Demand Curves

## Application of the Theory

Consider a cost-benefit analysis (CBA) of a two service (handbook and inquiry response) IAC. At issue is whether the existence of the IAC can be justified on economic grounds, and which services it should provide. Simply, does the value of all the benefits provided by the IAC exceed the costs of running the IAC? The problems traditionally encountered in such economic studies of information services are of three types:

- Identifying all, but only, the legitimate costs and benefits of the IAC;
- 2. Constructing conceptual measurement schemes:
- 3. Gathering the appropriate data.

We hope to show that, in an admittedly limited way, our theory helps mitigate these problems, including the last, by suggesting that under certain circumstances limited data is sufficient.

CBA of an IAC demands the introduction of several additional concepts: the social demand curve and the average cost curve. The social demand curve shows society's willingness to pay for each incremental quantity of information, where society includes the user. The importance of the concept stems from the recognition that there potentially exist positive externalities in the consumption of information i.e., not only do the direct users benefit but, through processes not completely understood, others eventually benefit as well. Thus, the social demand curve is always at least as great as the users' demand curve. Figure 8 illustrates these demand curves. The reader should bear in mind that the very existence of benefits over and above those accruing to the users is a matter of conjecture. No firm evidence of these benefits has yet been adduced. Without such evidence, the shape of the social demand curve is also a matter of conjecture. But standard economics suggests these external benefits be represented as diminishing with increases in total information supplied. D is the market demand curve, De the social demand curve.

Figure 9A illustrates the total cost curves for an IAC. TC represents the total costs of an IAC doing inquiry response alone, and TC<sub>H</sub> adds to that the (assumed) fixed costs of publishing a handbook. Assume the IAC information bases for the inquiry response and for the handbook are identical. Also, assume constant returns to scale in inquiry response.\*

Figure 9B displays the average cost curves derived from the total cost curves.

The complete model for the CBA of the two-service IAC is presented in Figure 10. We assume the unit price of inquiry response at the IAC is set at P. P may be set at marginal cost or any other value. The following analysis may be carried out with P a variable, if desired. This would be appropriate if the desire were to optimize some value, such as net social benefits, IAC net income, etc. To simplify exposition, however, we maintain P a constant here. Let us use the model to analyze the relative merits of four alternative situations:

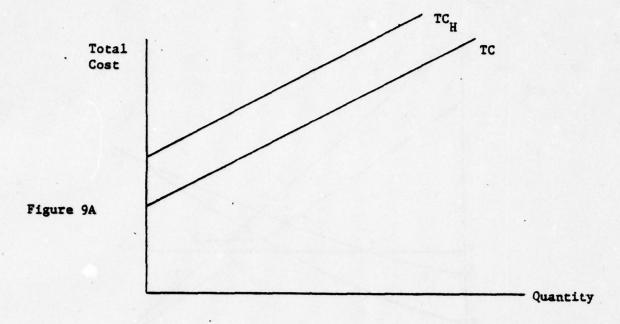
- 1. No IAC
- 2. IAC with only inquiry response
- 3. IAC with only handbook
- 4. IAC with both inquiry response and handbook.

Let us first analyze alternative #1. Using economic surplus as the value measure, the net benefits associated with the first possibility are area ABCE. This is the total surplus of <u>social</u> willingness to pay over costs incurred when Q' units of information are obtained. Q', of course, is the value determined by the intersection of S and D.

Now posit the existence of an IAC offering only inquiry response, priced at P per unit. Altogether Q'' units of information will be obtained, Q'' - Q''' from the IAC. The net benefits\*\* of this alternative are ALKE + LGQ''Q''' - ZYQ''Q'''.

<sup>\*</sup>By this meant that any change in output causes a proportional change in costs, e.g., doubling output doubles costs.

<sup>\*\*</sup>Since P may be arbitrary, the area under P is not necessarily a measure of real cost. Thus, gross benefits and gross costs correctly enter the expression.



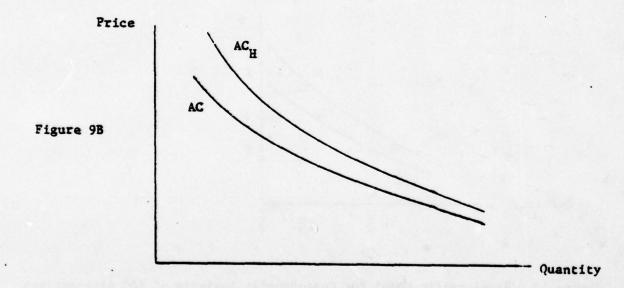


Figure 9. IAC Total and Average Cost Curves

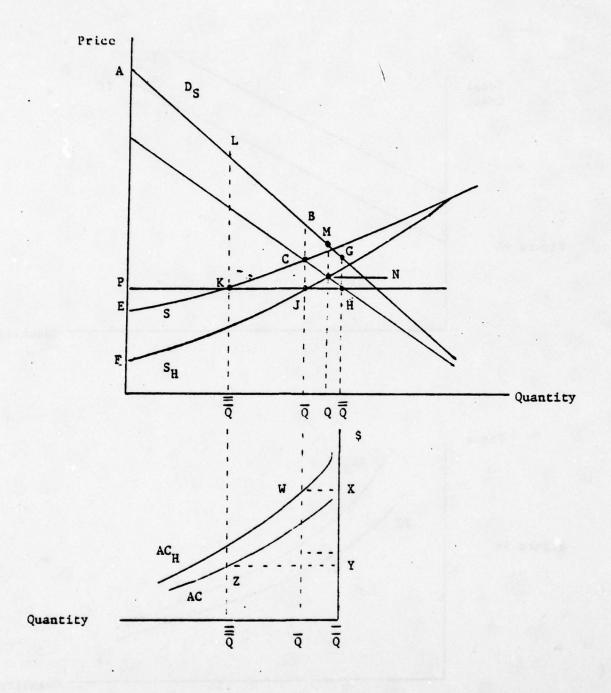


Figure 10. Diagiammatic Model for Cost-Benefit Analysis of IAC Alternatives

The net benefits associated with alternative #3 are AMNF - T, where T is the total cost of publishing the handbook alone, and Q is the quantity of information obtained.

The net benefits of alternative #4 are ABJF + BGQ''Q' - WXQ''Q'.

Clearly, the existence of the IAC is justified if any of alternatives 2, 3, or 4 have greater net benefits than 1. The preferred configuration for the IAC is given by the alternative (2, 3, or 4) with greatest net benefits. As might be expected, the issue can only be resolved by empirical analysis, and then on a case-by-case basis.

## Summary

An economic model has been constructed which appears to successfully account for a number of qualitative observations on the characteristics of the market demand for IAC services. The model shows that in the presence of competing information delivery systems, the IAC cannot expect the demand for <a href="its services">its services</a> to mirror the demand for information. The model provides a useful structure for analyzing various economic issues pertaining to IAC's, including viability, mix of services, and pricing. The paper concludes with a conceptual approach to cost-benefit analysis using the model.

A FORUM APPROACH TO SOFTWARE DEVELOPMENTTHE NATIONAL FORUM ON SCIENTIFIC AND TECHNICAL COMMUNICATION

by ETizabeth Byrne Adams
Department of Management Science
George Washington University
Washington D.C. 20052

Prepared for

The Software Life Cycle Management Workshop

Airlie, Virginia

August 1977

# A FORUM APPROACH TO SOFTWARE DEVELOPMENT THE NATIONAL FORUM ON SCIENTIFIC AND TECHNICAL COMMUNICATION

This paper will summarize the activities and findings of the National Forum on Scientific and Technical Communication, a fifteen month study supported by the National Science Foundation, Division of Science Information and conducted by the George Washington University, Science Communication Division. The Forum's relevance to software life cycle management lies in its: 1) evaluation of the current delivery systems for scientific and technical information (STI); 2) reliance on the users of STI for their suggested improvements to the delivery systems; 3) consideration to the current providers of STI and their problems; and, 4) the development of a model of an improved scientific and technical communication system and its presentation to management — who in this case are decentralized, autonomous and specialized. The report will focus on the: 1) methodology of the Forum; 2) findings; 3) advantages and disadvantages of a Forum approach; and, 4) future work of the Forum.

The availability and delivery of scientific and technical information in the United States is related to the quality of scientific research, the economy (as a result of technological developments), the competitive position internationally and the national security. The legislation that established the National Science Foundation included provisions for improving the interchange of scientific information among scientists in the United States and foreign countries. This

authority was considerably expanded in 1958 by direction to the Foundation to "... provide or arrange for the provision of indexing, abstracting, translating, and other services leading to more effective dissemination of scientific information" and "... undertake programs to develop new or improved methods, including mechanized systems, for making scientific information available."

The Division of Science Information (DSI), formerly the Office of Science Information Service, implemented this mandate through activities designed to improve the means for disseminating scientific information. Until 1974 this effort was toward strengthening and expanding science information activities of the professional scientific and technical societies. Since 1974, DSI funding has been for projects designed to advance the state of the art in science communication by producing generalizable results that can help improve methods of acquiring, transmitting, retrieving and using information in any scientific or technical area. However, in recent years, DSI has undergone budget reduction from \$14 million to \$5 million and staff decreases from 70 to 22.

During the same time period, private for-profit organizations developed to provide specialized services of acquiring, transmitting, retrieving and delivering scientific and technical information, off-the-shelf or customized. These services are supportive to or central components of the present categores of STI. These categories are:

1) the discipline-oriented systems of the professional societies - each concentrating on the systematic organization of knowledge in a

particular domain of basic science; 2) the mission-oriented information systems of the federal agencies, e.g. for astronautics in NASA, for atomic energy in ERDA, for medicine in the National Library of Medicine;

3) the specialized information activities of private institutions and of industry such as special libraries, information analysis centers, indexing and abstracting companies, data base services, etc. and, 4) the information services that are maintained by colleges and universities.

An indication of the size of the nation's scientific and technical information systems is shown by the measure of total resource expenditures. From 1960 to 1974, the GNP grew 177 percent, R & D funding increased 136 percent and scientific and technical communication resource expenditures increased substantially by an estimated 323 percent. Total communication resource expenditures were estimated at \$2.0 billion in 1960 and \$8.5 billion in 1974.

During this period of growth and expansion, numerous studies assessed the quality, quantity and direction of the scientific and technical communication systems. In most cases, policymaking recommendations resulting from these studies were virtually ignored. Meanwhile, significant changes have taken place in STI activities, which are indicative perhaps of greater change in the future. Recognition of these changes has been reflected in the National Science and Technology Policy, Organization and Priority Act of 1976 (PL 94-282). This Act established the Office of Science and Technology Policy (OSTP) directed by the President's Science Advisor and assisted by the President's Science Advisor Committee (PSAC), one of whose members to be an expert in

information dissemination. The Act also assigns to PSAC the task of conducting a survey of federal science and technology including the consideration of improvements for handling scientific and technical information in existing federal systems and in the private sector.

Among the changes which are significant to such a survey are:

- I. Users One estimate indicates that there will be a 20-30% increase in the number of scientists in the next decade. In addition, business, industry and governments are important secondary users of STI. Public concerns related to land-use planning, supersonic transports, ecology, pollution and the quality of life are engaging the time of private citizens, organized public and private groups, and state, local and federal government officials. Discussion of public policy issues such as these requires scientific and technical information accessible to a scientist in a specialized discipline or to one who doing interdisciplinary work or to a decision-maker/policy-maker or to a non-scientist citizen.
- 2. Providers The expansion and diversification of the "providers" of scientific and technical information to general groups of producers (generators, wholesalers, search and sell), providers (retailers, search and sell), and intermediaries (librarians, information resource specialists), has generated new industries and created new professional activities.
- 3. Requirements The value of STI solely for validation of research findings and minimization of repetitive efforts has been expanded to require STI for planning, policy-making, decision-making and the allocation of resources. However, cross-disciplinary information systems

integrating the diverse social and technical information bearing on problems such as the environment or energy have not been developed.

- 4. Technologies The capabilities inherent in data banks, abstracting, indexing and retrieval systems and expanded use of terminals permit increased access, inquiries which range from specific and detailed to general and macro-oriented and the ability to store massive amounts of information in large data banks. It was estimated that in 1977, there are more than 1,500,000 on-line computer terminals installed permitting wider access and alternatives for user-oriented formats.
- 5. Policy In recent years, there have been changes in federal policy in the areas of page charges, research direction, mailing rates, dissemination through NTIS and SSIE and information analysis centers. At the same time private industry has assumed new roles as providers of information.
- 6. Legislation Recent copyright legislation may resolve the problems concerning property and commercial rights as affected by photocopying and networking. Anti-trust laws which have the result of prohibiting cooperative arrangements among commercial organizations may help or hinder the user of STI services. The application of the Freedom of Information Act to publicly financed government data bases continues to be confusing.
- 7. International STI In recent years, a strong international scientific and technical communication system has evolved. International information transfer has added new dimensions to STI activities.

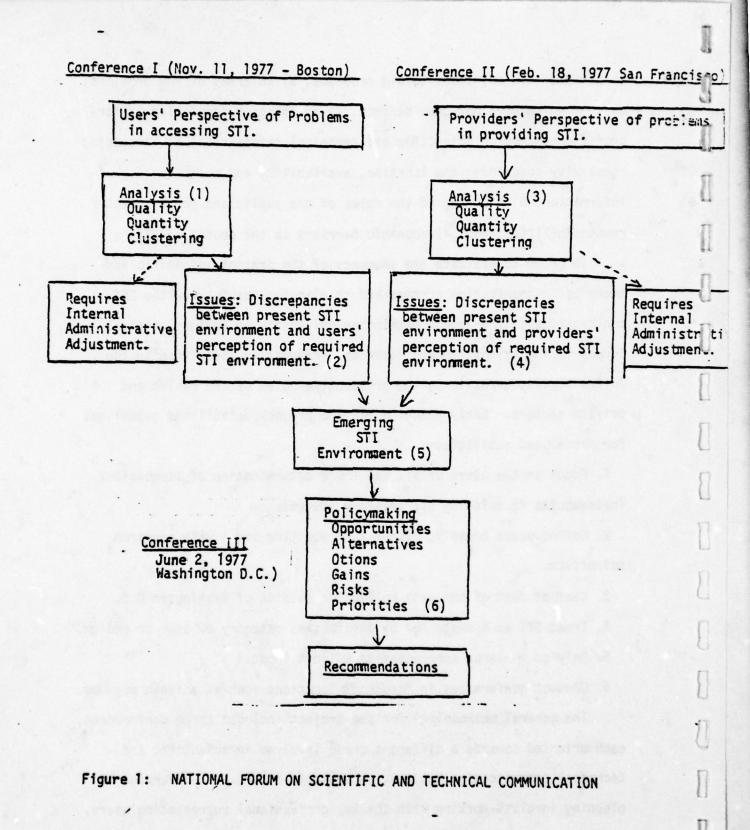
These factors serve to create: 1) a varied population of users; 2)

an array of new solutions to old problems; 3) an array of new problems, many of which have not been defined; 4) an emerging population of forprofit providers of scientific and technical information; 5) increasing complexity concerning the location, availability and accessibility of information; 6) blurring of the roles of the public and private sector responsibilities; and, 7) economic barriers to the access of STI.

In order to evaluate the adequacy of the present scientific and technical communication systems and to plan for the future, the DSI contracted with the George Washington University to explore critical issues facing scientific and technical information activities in the United States, emphasizing the prospective roles of the public and private sectors. Early planning for the project established guidelines for subsequent activities:

- T. Focus on the users of STI and their determination of adequacies/ inadequacies in existing STI delivery systems.
- 2. Define users broadly, rather then equating users with research scientists.
  - 3. Conduct most of the data collection outside of Washington D.C.
  - 4. Treat STI as a whole not by discipline, category of user or medium.
  - 5. Rely on workshop approaches to solicit input.
  - 6. Conduct conferences in "neutral" locations such as science museums.

The general methodology for the project included three conferences, each oriented towards a different group involved in scientific and technical communication systems, as shown in Figure 1. Conference planning involved working with the key professional representing users,



then, providers and finally policymakers. The first conference, the Users' Perspective Conference, focused on the user of scientific and technical information by addressing the question, "Are the users of scientific and technical information served adequately?" Professionals from public and private organizations', public interest groups and the press came together on November 11, 1976 at the Boston Museum of Science to discuss their viewpoints and perceptions about the availability and utilization of scientific information. The second Conference, the Providers' Perspective Conference was conducted at the California Academy of Science on February 18, 1977. It focused on the providers and/or intermediaries of scientific and technical information by addressing the question, "Are the providers of scientific and technical information adequately meeting the needs of the user?" On June 2, 1977, at the Smithsonian Institution, 168 participants at the Policymakers' Perspective Conference addressed the question: "Are there needs for public policies in the area of scientific and technical communication, and if so, what are the options?"

Participants at each Conference received, prior to the Conference, a background paper, a working vocabulary and a questionaire designed to stimulate workshop discussions. The collected questionaires and taped workshop discussions provided the record of all Conferences.

## Users' Perspective Conference

The Users' Perspective Conference was opened with a keynote address describing the changing context of STI. This was followed by four workshops constituting Session I. The material developed in the Session I workshops was summarized and presented to the assembled Conference.

The objective of Session I was to determine the critical problems and a ranking of importance of these problems, as perceived by Conference participants from their work activities in laboratories, research departments, political processes and other situations or environments where Conference participants use STI. Workshop size ranged from twenty to twenty-five people. The intent of these workshops was to stimulate thinking and identify problem relationships and subsets of problems from different perspectives. It was recognized that the problems of greatest significance to the output of the Conference were those which could be assisted by public policy action. However, the relationships between problem and policy was not always obvious, so policy action as a restraint was not placed on the discussion. Each participant was asked to bring to the Conference a completed questionaire on which they identified two critical problems they encountered and potential public policy actions to eliminate these problems.

The role of the presider at each workshop was: 1) Provide introductory remarks to the discussion; 2) Guide the discussion toward the desired output - seven critical problems; 3) Develop a summary of the workshop activities with the workshop assistant; and, 4) Present a fifteen minute summary to the assembled Conference.

The objective of Session II was to determine the responses, priorities, perceptions and recommendations by sector, to the problems and rankings developed in the workshops of Session I, and presented in the summarization to the assembled Conference. Session II workshops were organized by industrial participants, academic participants, government participants,

citizen groups and media. It was anticipated that the special needs, priority problem areas, relationships among problems and suggested public policy actions, perceived by sector, would provide useful comparative data.

The problems and issues raised in the sorkshop discussions may be categorized as those relating to:

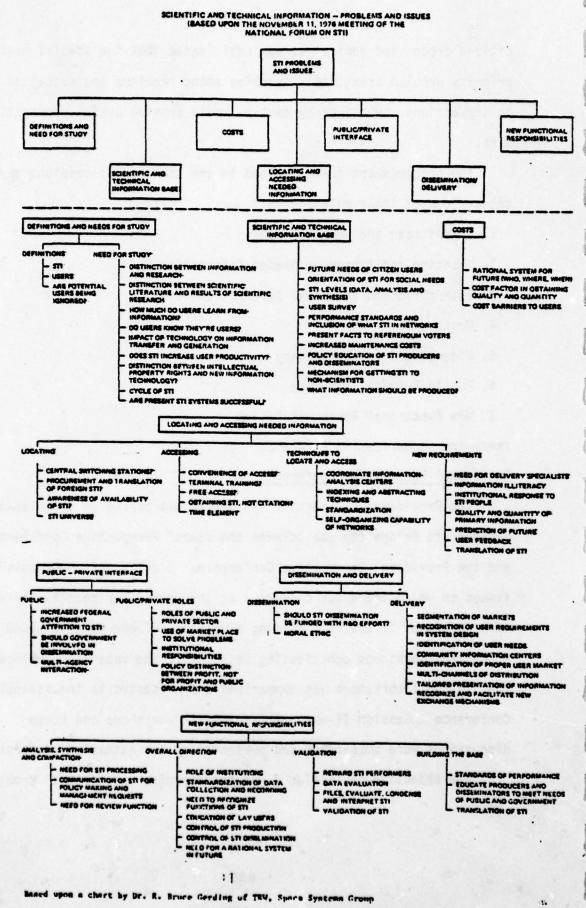
- 1. Definitions and Meed for Study
- 2. Locating and Accessing Needed Information
- 3. Scientific and Technical Information Base
- 4. Costs
- 5. Dissemination and Delivery
- 6. Public/Private Interface
- 7. New Functional Responsibilities

These are summarized in Figure 2.

## Providers' Perspective Conference

The Providers' Perspective Conference was opened by a series of talks designed to bridge the gap between the Users' Perspective Conference and the Providers' Perspective Conference. Significant problems and/or issues on which there was consensus at the Users' Perspective Conference were developed in the four opening addresses. These were followed by four concurrent workshops constituting Session I. The material developed in the Session I workshops was summarized and presented to the assembled Conference. Session II was composed of six workshops and these discussions were summarized and presented to the assembled Conference.

The objective of Session I was to determine the critical problems



and a ranking of importance of these problems, as perceived by Conference participants from their role as providers of scientific and technical information or intermediaries in the STI process. Participants in each of the workshops addressed the theme of the Conference: "Are the providers of scientific and technical information adequately meeting the needs of the user?" In doing so, the workshop served the dual role of responding to problems in accessing scientific and technical information as identified by users; and, identifying problems encountered by the providers of scientific and technical information. The Presiders guided the discussion toward a rank ordered list of seven most critical problems identified by workshop participants and presented a summary of the discussion to the assembled Conference.

The objective of Session II was to discuss the elements of each fissue area raised during the workshops of Session I. The Presider served the same role as in Session I. The issue areas which emerged from the problem analysis of Session I were:

- T. Economics costs, financing, cost barriers, budgets, mechanisms to equalize access, public/private good, new costing methods, basis research on the use of information.
- 2. Institutional Roles and Responsibilities centralization, decentralization, role definition, responsibilities, networking, international and inter-institutional cooperation, control, regulation, catalyst.
- 3. Information about Resources Available directory, central switching, selling skills, "glut".

- 4. Availability technical reports, unpublished information, access to governmental data bases, all non-classified data, document delivery, international document availability and source document provision.
- 5. Standards common command languages, structures, service (document delivery back-up), software, formats.
- 6: Delivery for Different Groups sophisticated/lay, foreign to English, inter-disciplinary, scientists and decisionmakers, link to public policy-making.
- 7. Education citizens, data inputters, retrievers, decision-makers, recognition of information costs, selling skills.

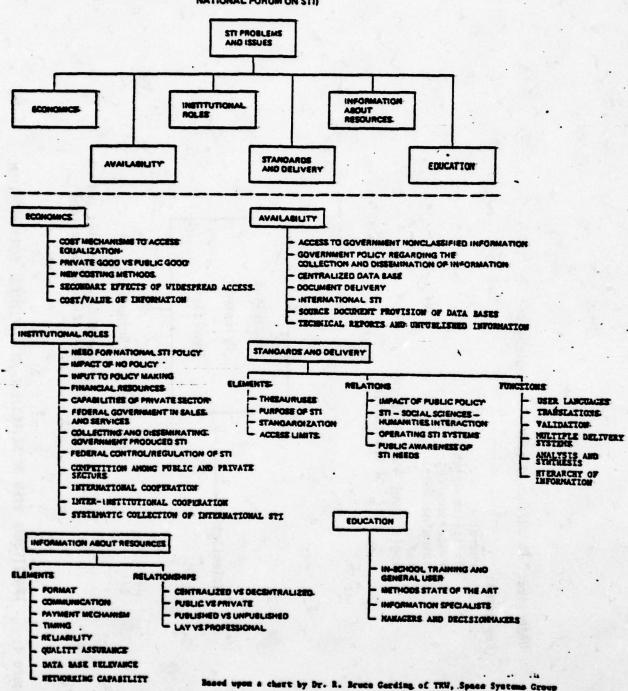
  The findings of the Providers' Perspective Conference are summarized as Figure 3.

The discussions at the Providers' Perspective Conference showed different emphases from those displayed at the Users' Perspective Conference. However, both perspectives displayed problems in using and providing STI in an environment which supports the traditional STI system as shown in Figure 4. In the traditional system, the scientist is the generator and user of disciplinary scientific information for the identification of research opportunities and the validation of research. The emerging perspective of a scientific and technical communication system was constructed to minimize the identified problems, issues and incorporate recommendations of the users and providers of STI. This is shown in Figure 5.

The emerging model of scientific and technical communication has two objectives - to advance scientific and technical knowledge and utilize

Figure 3: PROVIDER'S PERSPECTIVE

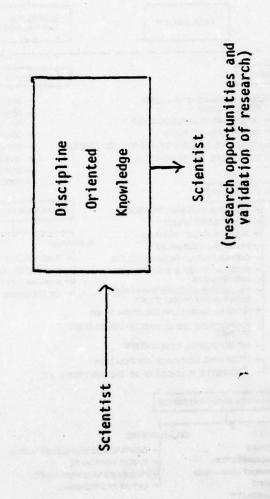
SCIENTIFIC AND TECHNICAL INFORMATION PROBLEMS AND ISSUES (BASED ON THE FEBRUARY 18, 1977 MEETING OF THE NATIONAL FORUM ON STI)



Objective: To advance scientific knowledge,

Key Terms:

Generator
Discipline-oriented
Primary sources
Dissemination
User
Single level - no hierarchy



TRADITIONAL FIEW OF SCIENTIFIC AND TECHNICAL COMMUNICATION Figure 4:

Total Control

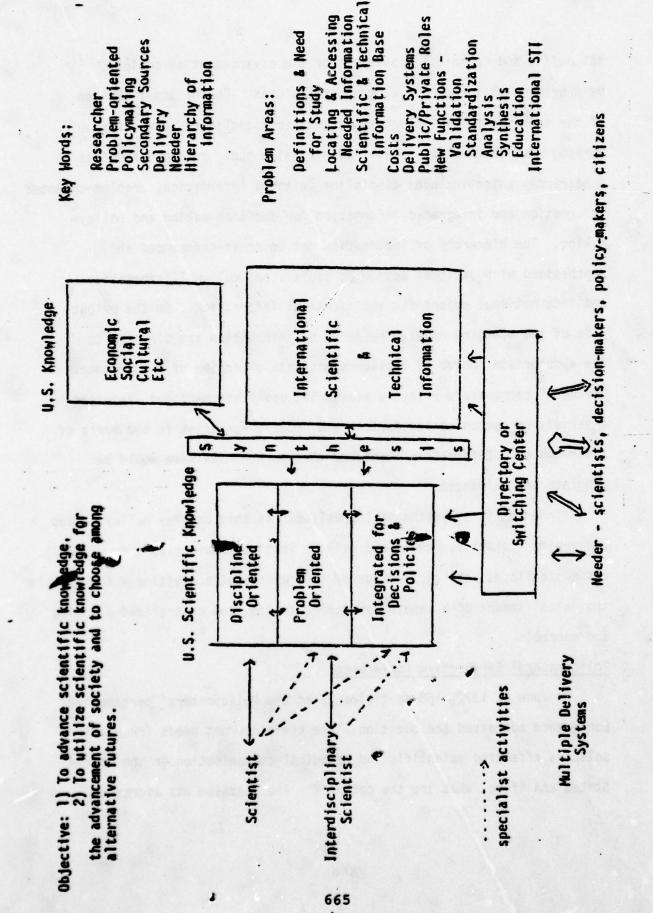


Figure 5: EMERGING MODEL OF SCIENTIFIC AND TECHNICAL COMMUNICATION

scientific and technical knowledge for the advancement of society

by providing choices among alternative futures. On the input portion

of the emerging model, scientists and interdisciplinary scientists

working with information specialists organize their research findings in

a hierarchy which includes discipline oriented information, problem-oriented
information and integrated information for decision-making and policy
making. The hierarchy of information may be cross-referenced and

synthesized with national economic, social, and cultural information

and international scientific and technical information. On the output

side of the emerging model, "needers" of information are directed to

the appropriate source of bibliographic data or copies of a reference.

"Needers" communicate with the stored STI using standardized languages,
a variety of communication devices and receive responses in the media of
their choice. Information furnished through the data base would be

complete and validated.

Inherent in a hypothetical model such as this are key policy issues concerning economics, public and private institutional roles and responsibilities, the performance of new functional activities - for example the establishment of a central switching system, and centralized planning and control.

## Policymakers' Perspective Conference

On June 2, 1977, 168 participants at the Policymakers' perspective Conference addressed the question: "Are there current needs for public policies affecting scientific and technical communication in the United States and if so, what are the options?" The question was addressed

via four workshops, each oriented toward an issue area defined by the previous two conferences. The issue areas were:

- I. Whether there is a need for economic mechanisms to equalize access to scientific and technical communication, to decrease costs, and to support research and development on the use and benefits of such information.
- 2. Whether institutional roles and responsibilities should be controlled by regulations in a decentralized or centralized manner in networking, inter—institutional cooperation and international arrangement, including the appropriate role for the private sector and intergovernmental science and technology sharing.
- 3. Whether the development and support of new functional activities for scientific and technical communication such as validation, standardization, analysis, synthesis, citizen education and a directory service or central switching system to information should be institutionalized and in which institution.
- 4. Whether centralized planning is required to manage the Nation's scientific and technical information resources to meet the different need levels and delivery requirements of the receivers who may be scientists, decision-makers, policy-makers or non-specialist citizens.

Preceding the workshops, each issue was addressed in a prepared statement by prominent individuals in information science and the associated issue area. Each workshop had an assigned Presider, Resource Person, and Assistant. The participants selected workshops in accordance with their interests. During the discussion, the Presiders were encouraged to

solicit comments oriented toward specific issues within the area and specific public policy options for each issue. A summary of the workshop discussions includes not specific policy choices but significant questions which need to be addressed prior to policy-making. A summary of the four issue area discussions follow.

#### 1. Issue - Eonomic Mechanisms

#### General Themes

-accuracy of information re: sources and users of STI -units of measurement for policy and valuation

#### Cost/Value Relationships

-cost of improvement vs exploitation of government STI -cost/value relationships in formal vs informal STI systems

#### Structural Issues

-relationship between STI policy and technology policy

-relative costs of formal vs informal STI systems.

-clarification and impact assessment of role and responsibilities of public and private sectors

-is STI naturally monopolistic

-cost structure for STI production and distribution

#### Societal

-should government ensure that an "informed" and honest market exists

-is it government responsibility for labeling STI products as "not necessarily certified"

-how far should the government go in translating STI for the commom citizen

-should government critically assess its own STI

-should STI be viewed as a universal input, requiring public utility regulation

#### Issue - Roles and Responsibilities

#### General Themes

-is this a market area or resource regulation area -identify focal points for national information policy

#### Procedural

- -standards and enforcement
- -international coordination
- -tax policies
- -validation
- -need for intermediary information gnerator synthesis
- -for whom should information be provided

#### Institutional

- -federal/state roles
- -different institutional roles
- -sensitivity to constitutional, moral, political restraints
- -competition between private and public

#### Demand Orientation

- -STI for different publics
- -state and local needs
- -policymaker needs
- -capacity to use
- -new functions role of essayist, book/article review and validation
- -purpose of information
- -different kinds of STI

#### 3\_ Issue - New Functions

#### General Themes

- -lack of attention, interest and focus on information. Information policy is ad hoc. Need a set of assessment and value criterija for information. Policy should address access, announcement, investment, documentation. What are driving forces in STI?
- -public relations activities required demonstrations effort on communications technology, national effort dealing with education curricula.

#### **Issues**

- -develop numeric data for recording locations, promotion of data centers -coordinate systems, data bases and networks
- -communication within information science field promote, examine and test new approaches and demonstrate
- -improve education re: information access for all groups and levels of users
- -roles and technologies in providing information use dynamic there isn't a model of a future "system" to serve as a guide for the present

-STI transfer might be resolvable through information science and technology - requires research and communication between scientist and information specialist.

-quality of STI - ascertained through screening and filtering devices

#### 4. Issue - Management of STI

#### General Themes

-information is a resource requiring appropriate management, not ad hoc, fragmentary, sporadic

-government should exert leadership in establishing a model system with a single classification, handling, processing and access methodology to serve disciplinary, interdisciplinary and policy research

#### Issues

-manage information as a resource as we manage human, financial and other resources

-reduce barriers to sharing information underlyine public policies, science and technology

-improve federal government information management and coordination and standardization processes

-orient STI systems to user demand

-perform R & D to improve STI interface between sources (producers) and users

-develop explicit federal policy on import/export of STI

-improve foresight and planning for crisis management through
STI improvement

-develop management techniques to expand the narrow parochial channels which restrict the flow of STI and its utilization

-disseminate products of federal R & D expeditiously

-finance communication of STI adequately as independent function

## Conferences' Analysis and E valuation

The activities of the Forum at this time involve synthesis and analysis of the problems and issues described in this report and the approximately 500 questionaires collected during the three conferences. The Forum Report, which will be completed September 30, 1977 will include a hierarchy of problems, issues requiring more research and some specific policy recommendations.

The following are some general observations based on the conferences and associated activities.

- I. There is wide interest in scientific and technical communication as shown by the conference registration figures, travel involved in attending conferences, a one day investment of time, active workshop discussions and detailed answers to the questionaires.
- 2. There are needs for public policy in scientific and technical communication as expressed at all the conferences. In particular, the Users' Perspective Conference was characterized by appreciation and enthusiasm for "their" opportunity to express their needs.
- 3. Public policy recommendations were difficult to specify. In all discussions, there appeared to be a sense of "the whole is bigger than the parts" and none of "us" see the whole. Unwanted byproduct effects of premature policy recommendations introduced a reluctance to state recommendations.
- 4. There appeared to be much support for the view that scientific and technical communication is an entity requiring management, planning and a future orientation as opposed to being simply a by-product of another managed process (R & D funding).
- 5. Scientific and technical communication has become more closely related to other information disciplines and "national information policy" considerations then it was previously.
- 6. That the federal government should exert greater leadership in coordinating its own scientific and technical information systems, developing model systems and interfacing its activities with the

private sector.

- 7. Recognition that scientific and technical communication is not now a politically valuable issue area.
- 8. Interest in greater sharing of STI among federal, state and local government jurisdictions may provide greater political interest and thus momentum to improvements in the nation's scientific and technical communication capabilities.
- 9. There are many "players" with vested interest in scientific and technical communications, whose contributions, roles and reactions must be appreciated and anticipated in policy-making considerations.

  Forum Approach

The Forum approach to software development is advantageous because it permits one to go to the "grass roots" level for data collection. In addition, when the federal government is involved, it is valuable to study any existing system, as it is perceived by citizens outside the federal bureaucracy. And, as their attention and views are expressed, their interest and support to the project may be won. Problem definition and criticality are perceived differently by different "players" in the hierarchy of any system. It is useful to analyze these different perceptions to clarify significant problems, as measured by established criteria. The conference format permits one to informally assess the different "players" in the process and define the power structure.

There are some disadvantages to a Forum approach, some of which may be overcome. Firstly, it is difficult tobe certain that the conference is attended by a representative sample of the population one is

interested in. This is particularly true when conferences are supported by federal money and an open door policy must prevail. At such meetings, specialized groups may overpower discussions and distort the findings. Full attention to conference activities and inattention to the operations of large institutions that are involved may preclude the detection of "behind the scenes" activities. These may neutralize or sabotage the outcomes of the conference.

Finally, there are certain ingredients which contribute to a Forum's success. These are:

- 1. Identification of knowledgable participants.
- 2. Appropriate timing for conferences and Forum recommendations.
- 32 Commitment of an organization to the Forum study.
- 4. Willingness to accept findings in return for participants' time investment.
- 5. Screening devices of input from vested interests.
- 6. Objective environment and conduct of the Forum activities.
- 7. Support of the power structure.

#### ATTENDEES

Dr. Elizabeth I	Byrne Adams	
11112 Corobon Lane		
Great Falls, V.	A 22066	

(202) 676-9070 (Work) (703) 430-1641 (Home)

Joel Aron
IBM Europe
Tour Franklin, CEDEX II
Paris, France

Telex 630096

Dr. Victor Basili Department of Computer Science University of Maryland College Park, MD 20742 (301) 454-4251 (Work) (301) 552-9721 (Home)

L. A. Belady Thomas J. Watson Research Center P. O. Box 218 Yorktown Heights, NY 10598 (914) 945-2825

Dr. Barry Boehm TRW Space Systems and Energy Group One Space Park Redondo Beach, CA 90278 (213) 535-2184

John Carrow General Electric 1755 Jefferson Davis Hwy. Suite 200 Arlington, VA 22202

(703) 979-6000, x345

LTC John P. Cover DALO, Rm. 2C574 Pentagon Washington, D.C. 20310

(202) 695-3069

Dr. George Cowan Computer Sciences Corp. 6565 Arlington Blvd. Falls Church, VA 22046 (703) 533-8877

Edmund B.	Daly	7
GTE Auton	natic	Laboratory
Northlake,	ILL	60164

(312) 562-7100, x2239

Dr. Thomas De Lutis Systems Research Corp. 5205 Leesburg Pike, Suite 505 Falls Church, VA 22041 (703) 379-6844

Dr. Bryce Elkins Computer Sciences Corp. 400 Army-Navy Dr. Arlington, VA 22202 (703) 521-5280

I. R. Elliott
Midland Bank
Griffin House (Pennine Centre)
41 Silver StreetHead
Sheffield S1 3GG
England

(Sheffield 20999X8201)

Major Ely 730 Peachtree Street, N.E. Suite 900 Atlanta, GA 30308 (404) 881-7461

Dr. Philip Enslow School of Computer and Information Science Georgia Institute of Technology Atlanta, GA 30332

(202) 921-3491

Dr. Dennis Fife Technology A 367 National Bureau of Standards Washington, D.C. 20234

Prof. George Fix Mathematics Department Carnegie-Mellon University Pittsburg, PA 15253

William S. Franklin	(202) 697-8636
Office Secretary of Defense (Comptroller)	
Room 1A658	
Pentagon	
Washington, D.C. 20301	
Dr. Maurice Halstead	(314) 493-3326
Purdue University	
West Lafayette, IN 47907	
Maj. Kenneth Hamilton	
College of Industrial Management	
Georgia Institute of Technology	
Atlanta, GA 30332	
Kenneth Kolence	(415) 493-0300
Institute for Software Engineering	
P. O. Box 637	
Palo Alto, CA 94302	
Dr. Phillip Korff	(703) 533-8877
Computer Sciences Corp.	
6565 Arlington Blvd.	
Falls Church, VA 22046	
Prof. Meir M. Lehman	01-589-5111 x2718
Department of Computing and Control	Telex 261503
Imperial College of Science and Technology	
180 Queen's Gate, London SW7 2BZ	
England	
Prof. Bev Littlewood	01-253-4399 x646
Department of Mathematics	
City University	
St. Johns Street	
EC 1V - 4PB	
London, England	
Dr. Thomas Love	(703) 979-6000 x344
General Electric Co.	

1755 Jefferson Davis Hwy.

Arlington, VA 22202

Suite 200

Dr. John H. Manley	(301) 953-7100
Johns Hopkins University Applied Physics Lab.	
Johns Hopkins Road	
Laurel, MD 20810	
Dr. Clement McGowan	(617) 890-6900
Softech Inc.	
460 Totten Pond Road	
Waltham, MA 02154	
Robert McHenry	(201) 840 7000
IBM Federal Systems Division	(301) 840-7232
18100 Frederick Pike	
Gaithersburg, MD 20760	
Carthersburg, MD 20100	
Dr. Clair Miller	(703) 790-3000
Honeywell Information Systems	
7900 Westpark Drive	
McLean, VA 22101	
J. D. Musa	(201) 386-2398
Bell Telephone Laboratories	
600 Mountain Avenue	
Murray Hill, NJ 07974	
Janet Nauta	(703) 521-5280
Computer Sciences Corp.	(100) 021 0200
400 Army-Navy Dr.	
Arlington, VA 22202	
	1986/67
Dr. Peter Norden	(914) 696-3183
IBM Corp.	
1133 West Chester Ave.	
White Plains, NY 10604	
Dr. Francis N. Parr	01-589-5111, x2729
imperial College of Science & Technology	Telex 261503
180 Queen's Gate	16167 701302
London SW7 2BZ	
England	
John K. Patterson	(314) 268-2972
Army Logistics Management	
Systems Agency	
210 N. 12th Blvd., P.O. Box 1578	
St. Louis, MO 63101	

John N. Postak	(301) 424-0270
Doty Associates, Inc.	
416 Hungerford Dr.	
Rockville, MD 20850	
Lawrence H. Putnam	(703) 979-6000
General Electric Co.	
1755 Jefferson Davis Hwy.	
Suite 200	
Arlington, VA 22202	
Andrew Reho	(703) 533-8877
Computer Sciences Corp.	1000
6565 Arlington Blvd.	
Falls Church, VA 22046	
Dr. J. S. Riordon	(613) 231-3625
Systems Engineering & Computing Science	
Carleton University	
Colonel By Drive	
Ottawa, Canada KIS 5B6	
David G. Robinson	(617) 862-8820
DP Management Corp.	talestati alba
1 Militia Drive	
Lexington, MA 02173	
Dr. Peter Sassone	(404) 894-2600
School Industrial Management	(101) 001 2000
Georgia Institute of Technology	
Atlanta, GA 30332	
Jules Schwartz	(213) 678-0311
Director, Senior Technical Staff	(213) 010-0311
Commercial Division	
Computer Sciences Corp.	
650 N. Sepulveda Blvd.	
El Segundo, CA 90245	
Li behando, en vonto	
Prof. Martin Shooman	(212) 643-5000
Polytechnic Institute of New York	(222) 010-0000
333 Jay Street	
Brooklyn, NY 11021	
2	

Dr. Dick Simmons Director Data Processing Center	(713) 845-4211	
Texas A&M University		
College Station, TX 77843		
College Station, 1A 11045		
Dr. John Staudhammer	(919) 782-4571	
348 Buncombe St.		
Raleigh, NC 27609		
Kenneth R. Stewart	(301) 424-0270	
Doty Associates, Inc.		
416 Hungerford Dr.		
Rockville, MD 20850		
Capt. Alan Sukert	(315) 330-2204	
Information Sciences Division		
Griffiss AFB, NY 13441		
Dr. Claude Walston	(301) 897-3009	
IBM Federal Systems Division		
10215 Fernwood Rd.		
Bethesda, MD 20034		
Capt. Thomas E. Watkins		
AFDSC/DMT		
Gunter AFB, AL 36114		
Raymond W. Wolverton	(213) 535-2619	
Office of Software Research & Technology		
TRW Defense and Space Systems		
Redondo Beach, CA 90278		
Prof. Marvin Zelkowitz	(301) 454-4251	
Department of Computer Science		
University of Maryland		
College Park, MD 20742		

## SOFTWARE LIFE CYCLE MANAGEMENT WORKSHOP August 21-23, 1977

Conference Coordinator: Dr. Bryce Elkins Workshop Directors: Prof. M. M. Lehman Col. L. A. Putnam

#### August 21, 1977

5:00 p.m. Social Hour

6:00 p.m. Dinner

7:00 p.m. Staff meeting

### August 22, 1977

7:30 a.m. Breakfast

8:00 a.m. General Session

9:15 a.m. Technical Sessions

10:45 a.m. Coffee Break

11:00 a.m. General Session

12:00 noon Lunch

1:00 p.m. Technical Sessions

3:00 p.m. Coffee Break

3:15 p.m. General Session

5:00 p.m. Social Hour

6:00 p.m. Dinner

7:30 p.m. General Session

After Dinner Address, Dr. Staudhammer

AD-A053 014

COMPUTER SCIENCES CORP ARLINGTON VA

SOFTWARE PHENOMENOLOGY - WORKING PAPERS OF THE SOFTWARE LIFE CY--ETC(U)

AUG 77 B ELKINS, L HUNT

DAHC26-76-D-1006

NL

UNCLASSIFIED

8 OF 8 ADA 053014

> ### ###



END DATE FILMED

## August 23, 1977

7:30 a.m.

9:00 a.m.

10:45 a.m.

11:00 a.m.

12:00 Noon

1:00 p.m.

2:00 p.m.

3:00 p.m.

4:30 p.m.

Breakfast

**Technical Sessions** 

Coffee Break

General Session

Lunch

**Technical Sessions** 

Plenary Session

Coffee Break

Closing Remarks